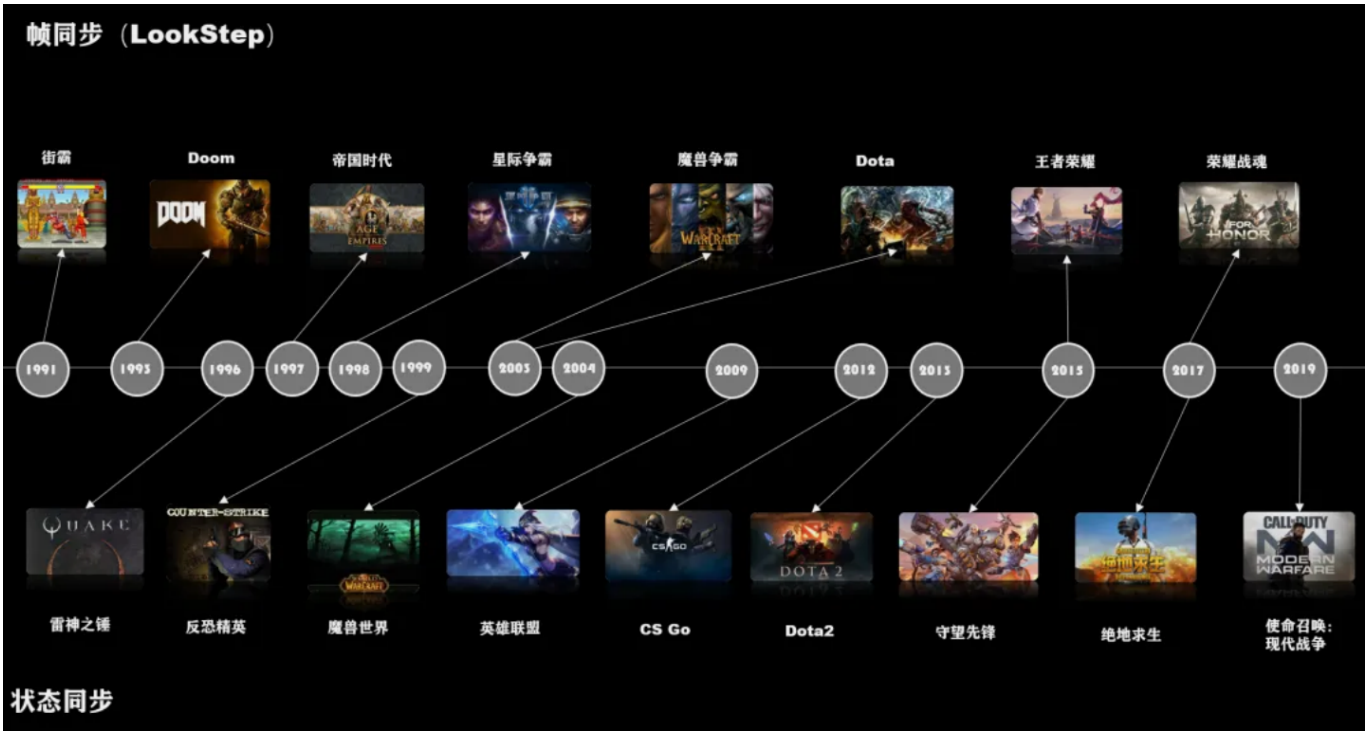
网络游戏的历程

https://www.bilibili.com/video/BV1ft4y147si/?spm_id_from=333.788.recommend_more_video.3&vd_source=4ef13c12c37e96927eed265ec739144b

网络游戏其实发展史也是短短几十年，从最早的快照同步到帧同步，再到状态同步，对于不同的游戏，采取不同的同步方式，甚至是复杂的混合同步。



局域网游戏

《帝国时代》、《CS》等等都有局域网对战的模式。它的网络拓扑结构多数时候是 P2P ，下文会提到。

多人游戏

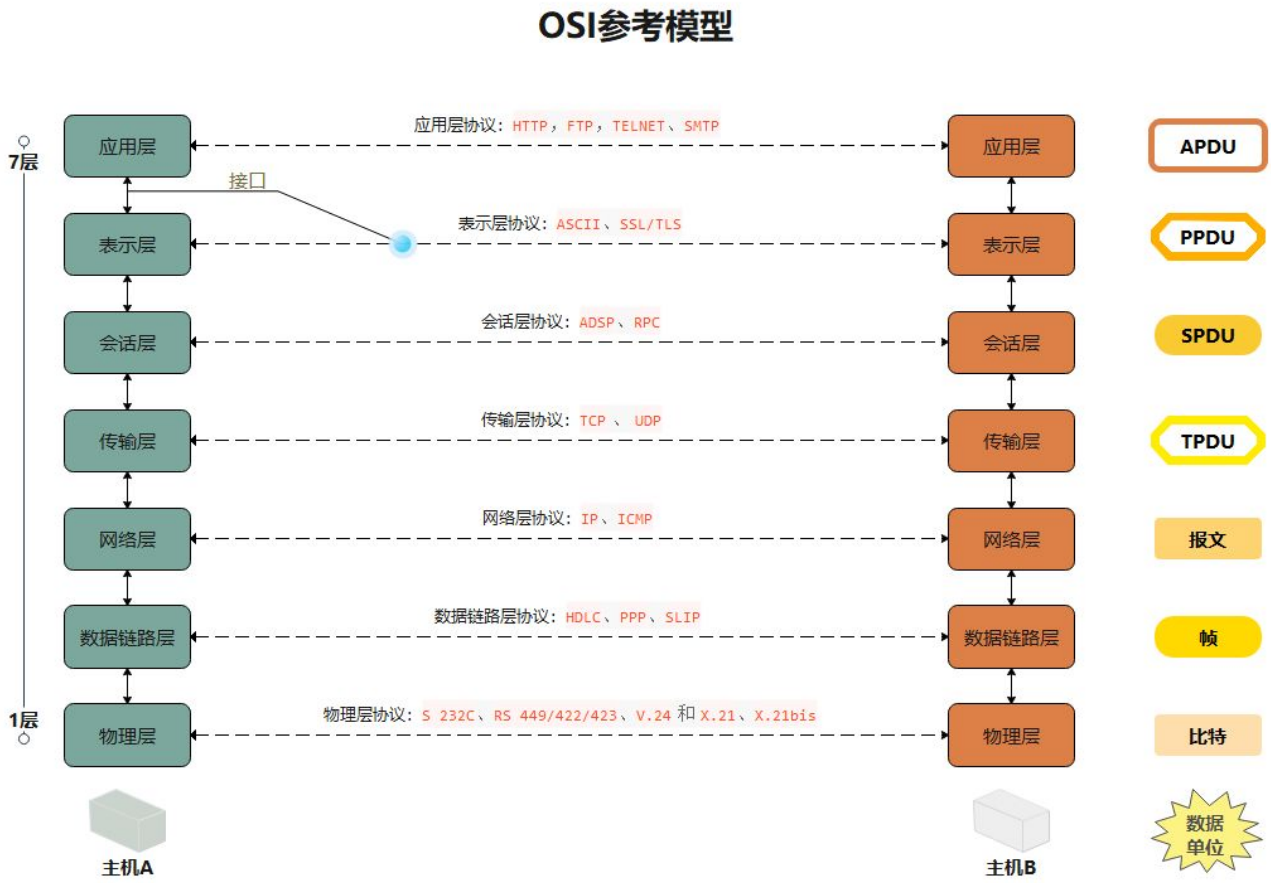
业内主流多人游戏类型包括：

- 大规模多人在线游戏（MMO）
- 弱交互回合游戏（例如：棋牌）
- 要求高及时性的动作、FPS、MOBA 等等游戏（例如：王者荣耀）

游戏同步的基本概念

多人网络游戏是电子游戏中一个极为重要的游戏品类，随着技术的发展，游戏服务器的架构发展与常规的 Web 服务器机构有了很大区别。在应用场景上，多人网络游戏，更加重视数据的即时性，我们需要玩家的输入，快速的表现在游戏中，例如玩家进行的移动输入，游戏画面就应当快速进行反馈表现。而常规的 Web 应用就不一样，及时性要差很多。

对于及时性、以及可靠性的不一样要求，往往就会有不同传输层的选择，如下图，传输层通常包含：TCP、UDP。



传输层协议的选择

TCP、UDP 是属于传输层，我们游戏选择的是 TCP，目前来说对速度要求不高，基本可以满足需求。

TCP

TCP 主要包含以下特点：

- 保证有序；
- 丢包重传；
- 传输速度较慢，这个源于慢启动、拥塞控制等原因引起速度慢，同时也是这样原因去保证TCP的可靠。

UDP

UDP 主要包含以下特点：

- 不保证顺序；
- 不主动重传、不关心是否对方接收；
- 传输速度快，因为不关心发送结果，没有拥塞控制等，所以传输更加没有约束，反而速度快了，及时性也就高了；

可靠 UDP

对于可靠UDP，我们可以 DIY 速度、可靠 方面的平衡，某些情况要求可靠，某些方面要求速度。

KCP 就是一个可靠UDP的传输协议。KCP是一个很简单的ARQ的实现，包括选择重传和快重传等机制，对上层提供一个可靠的字节流。应用层可以使用多流复用的框架来实现对多个流的支持。另外，KCP增加了可配置启用的加密和FEC选项，FEC用的是Reed-Solomon纠删码，例如可以配置发送10%的冗余数据，来减少丢包时需要的重传，从而降低数据传输的延时。

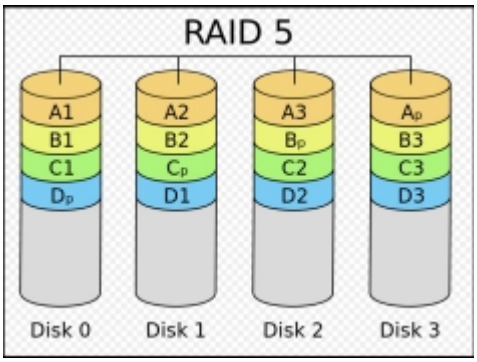
FEC （Forward Error Correction）

FEC：Forward Error Correction，前向纠错，FEC 是一种通过在网络传输中增加数据包的冗余信息，使得接收端能够在网络发生丢包后利用这些冗余信息直接恢复出丢失的数据包的一种方法。

XOR（异或）

```
0 ^ 0 = 0
1 ^ 1 = 0
0 ^ 1 = 1
1 ^ 0 = 1
```

硬盘中有一种数据的保护方式，通过存储冗余信息防止数据丢失， RAID 5 是一种磁盘阵列的方式（Redundant Array of Independent Disk），至少用三个硬盘，其中牺牲一些存储用来存储其他剩余的数据 XOR 码。



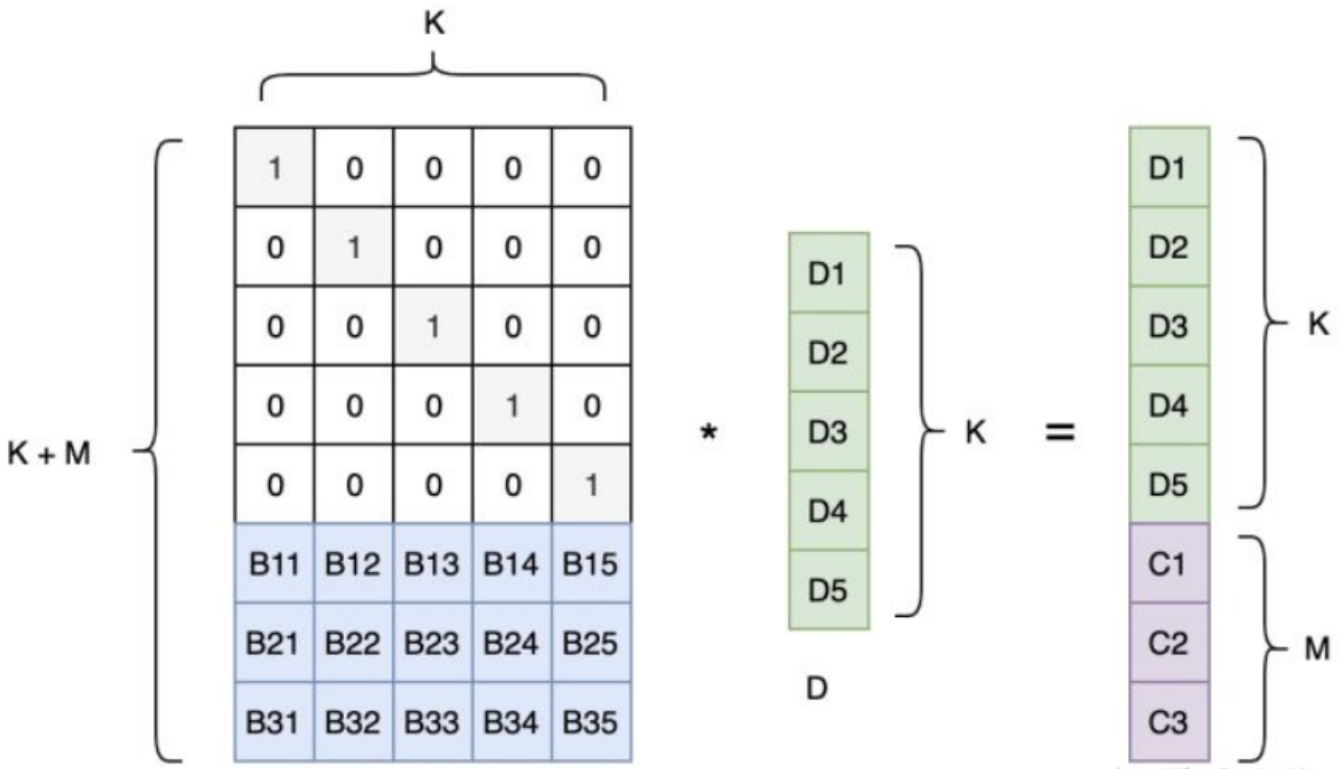
如上图所示，4个Disk 分别牺牲了一小块的区域，存储着 XOR 后的结果，这样如果一块 Disk 数据丢失，那么也可以通过 XOR 码 去恢复数据。

不过缺点也没明显，浪费了一定比例的存储空间，另外如果损坏的是两个Disk，那么数据将无法恢复。

同理于硬盘中的XOR使用，网络传输中 UDP 可以将传输的多个包进行XOR，并且也放入到传输包中，那么也可以和硬盘的恢复方式一样，但是缺点也是一样的。

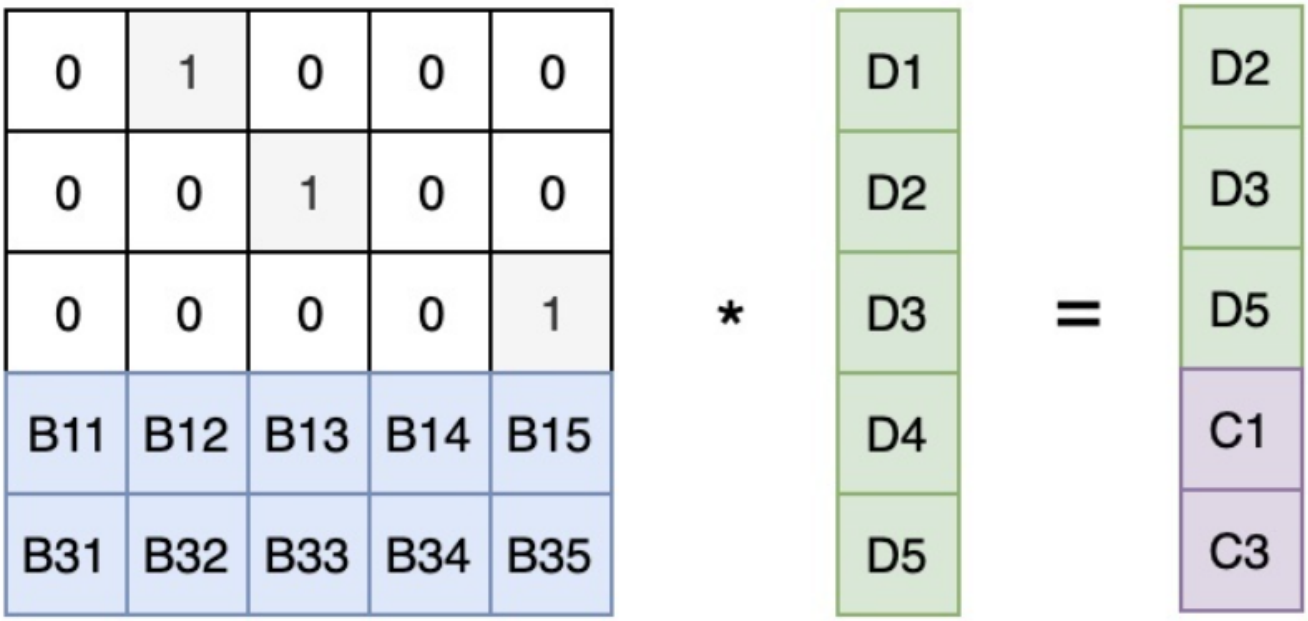
Reed-Solomon Codes

里德-所罗门码（Reed-solomon codes，简称 RS codes）。如下图，我们把数据拆成 K 份，M 就是增加的数据块，M块区域的矩阵唯一的要求就是保证矩阵的可逆性。

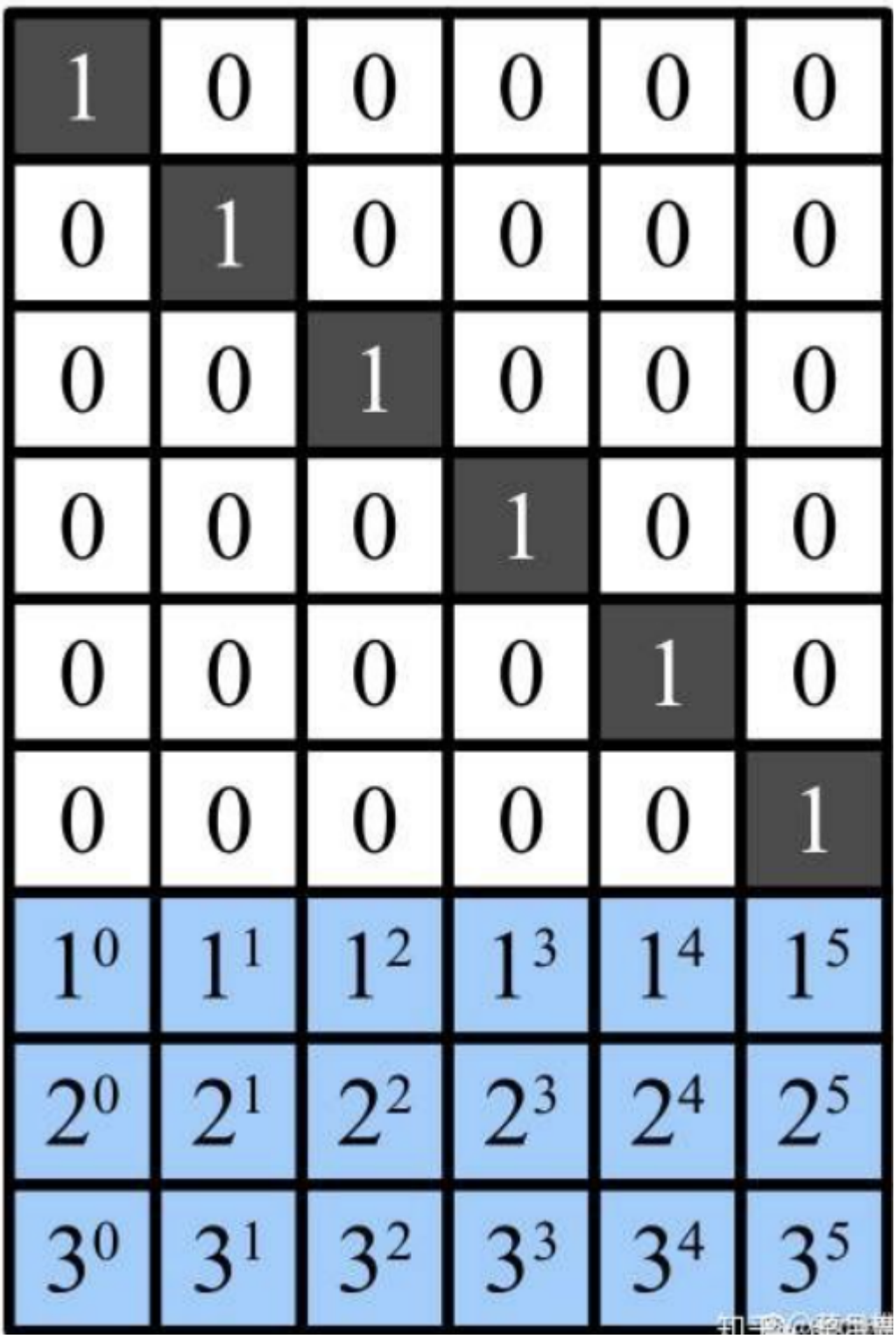


解码过程如下，丢失了D1、D4、C2。如果给下面的矩阵命名为X、Y、Z，下图表示的X * Y = Z。当前我们已知 X 、Z, 那么我们只需要求解 Y ，就算是恢复了丢失的数据。

根据矩阵运算公式，Y = X 的逆矩阵 * Z，最后 Y 就解出来了。



X 矩阵必须保证 任意行组成的矩阵具有可逆性。如下组成，具备这一特性。



网络拓扑

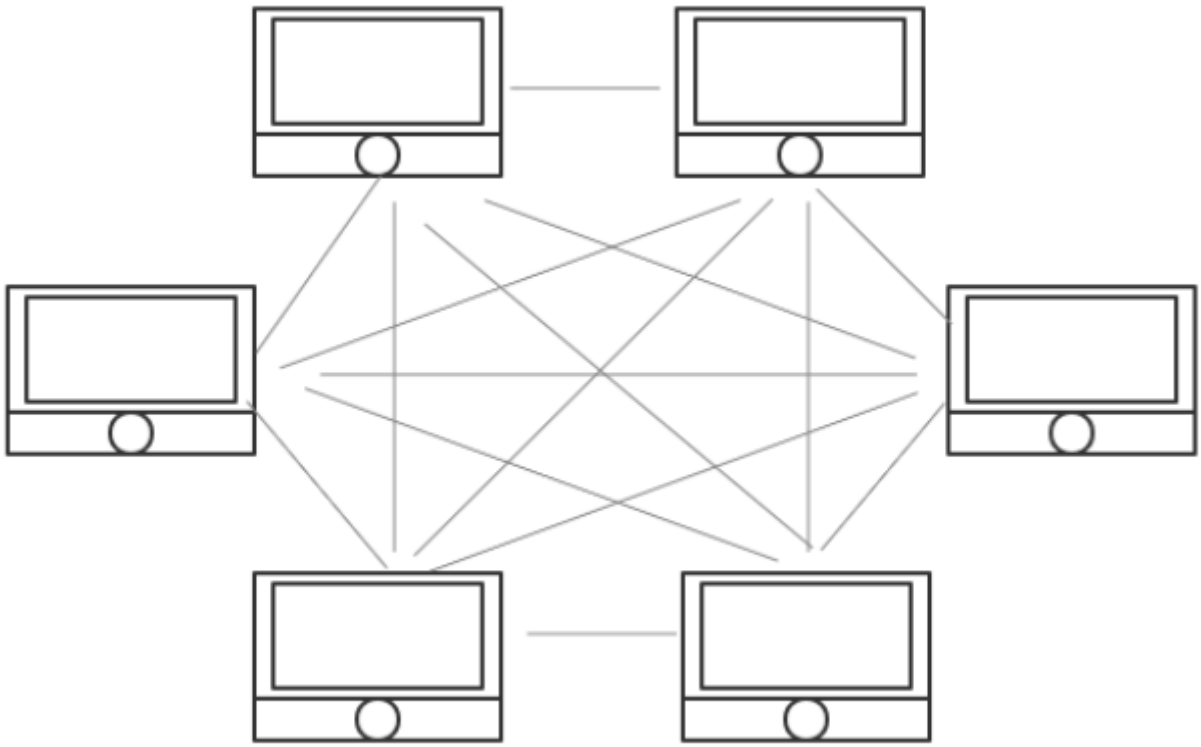
通常情况下，游戏同步强调的是数据的同步，当数据层的数据同步了，那么显示层就可以通过一些手段来做到同步。数据层同步，再驱使表现层同步。

不同游戏可以选择不同的网络拓扑结构，如果选择的网络拓扑结果不合适，可能会造成网络带宽的严重浪费。必要时候，逻辑运算是否在Server、Client上运行，也是值得考量的事情，过多消耗服务器算力，会造成机器成本过高，闲置客户端算力，也是一种浪费。

下文介绍一下主要的网络拓扑：P2P、CS。

P2P

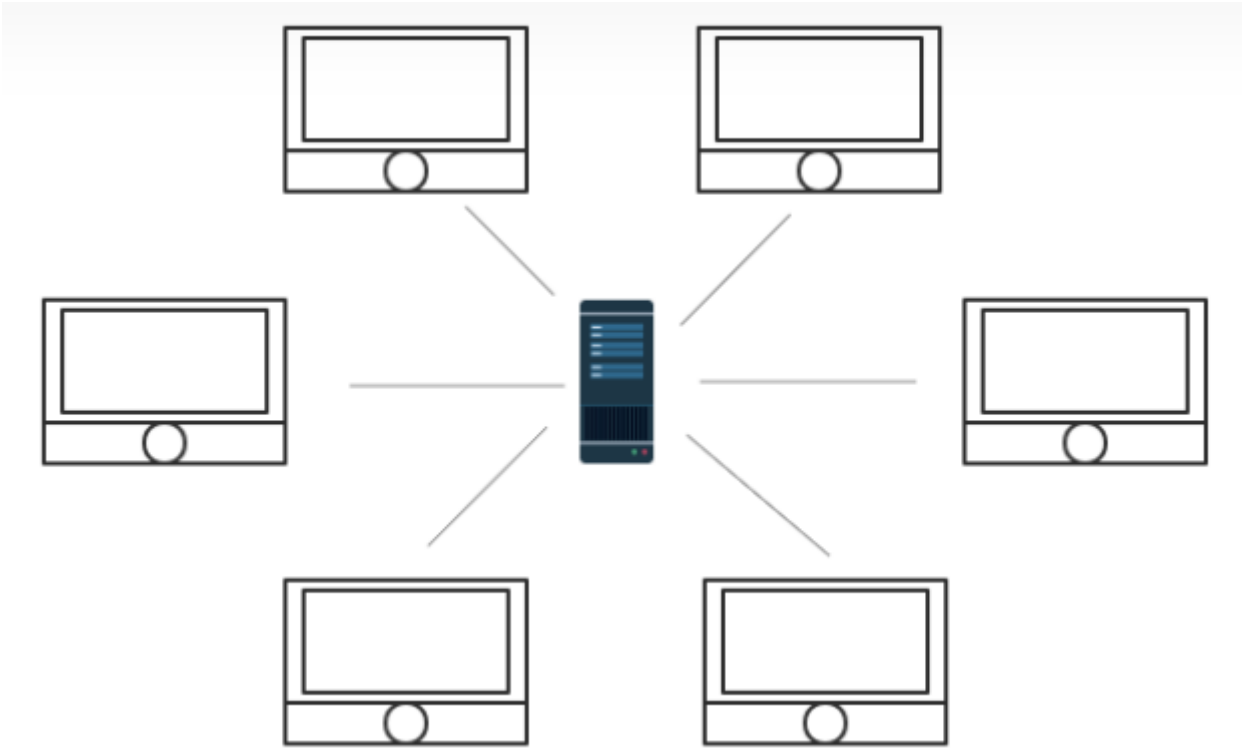
1973年夏天，高中暑期实习生在美国加利福尼亚州NASA的研究中心首次撰写了游戏《迷宫战争》[1]。通过使用串行电缆连接两台算机，增加了两人游戏功能。由于涉及两台对等的计算机，可以使用相同的格式化协议包相互发送信息，因此可以认为这是第一个P2P架构的电子游戏。在那个时代，并没有多人在线游戏，互联网也没有诞生，网络同步一词更是无人知晓。不过当两台计算机上的数据进行传递时，最最最简单的同步模型就已经悄无声息的出现了，A把操作信息通过电缆发给B， B收到数据后处理，在把自己的操作数据通过电缆发送给A。比较有意思的一点是，计算机技术的发展或多或少都与电子游戏有着紧密的联系，甚至很多技术的诞生就是源于对游戏交互方式的探索。



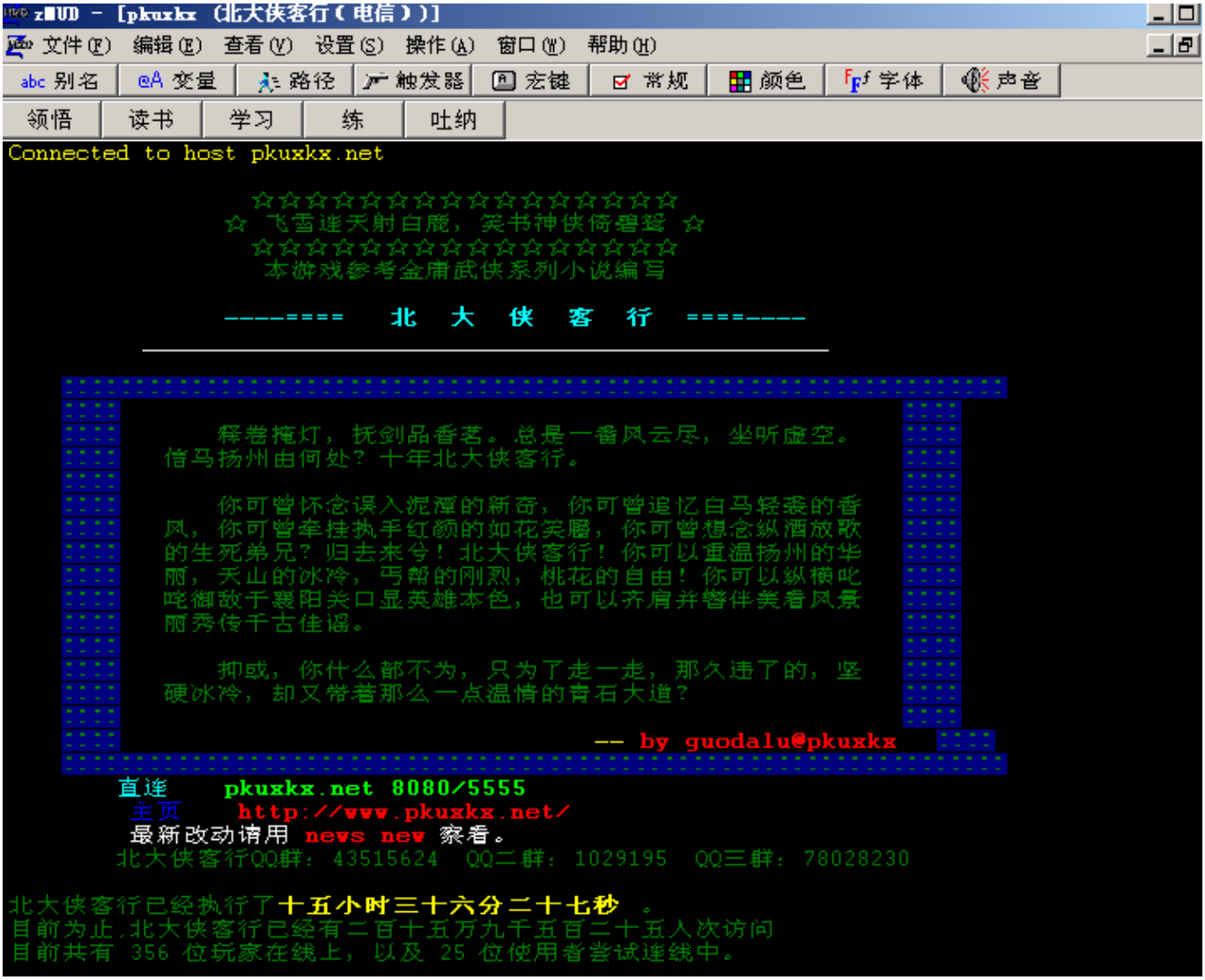
https://baijiahao.baidu.com/s?id=1689217666386425537

CS

1978年，Roy Trubshaw 编写了世界上第一个MUD程序《MUD1》，后来又在此基础上诞生了开源的MudOS（1991），成为众多网游的鼻祖。MUDOS使用单线程无阻塞套接字来服务所有玩家，所有玩家的请求都发到同一个线程去处理，主线程每隔1秒钟更新一次所有对象。这时候所谓的同步，就是把玩家控制台的指令发送到专有的服务器，服务器按顺序处理后再发送给其他所有玩家（几乎没有什么验证逻辑），这是最早的CS架构。当时PC图形化还不成熟，MUD早期的系统只有着粗糙的纯文字界面，由于也没有物理引擎等游戏技术，所以对网络延迟、反馈表现要求并不高。

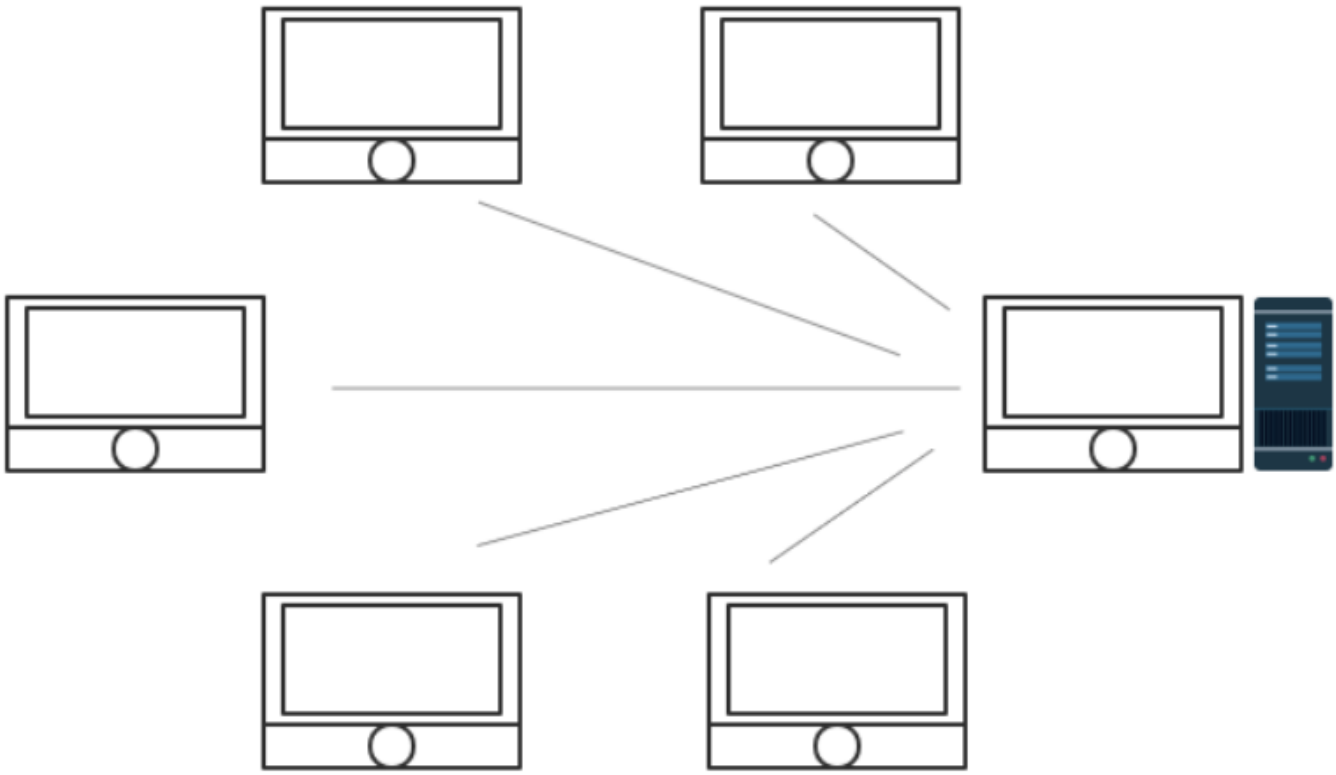


国产MUD游戏，《北大侠客行》



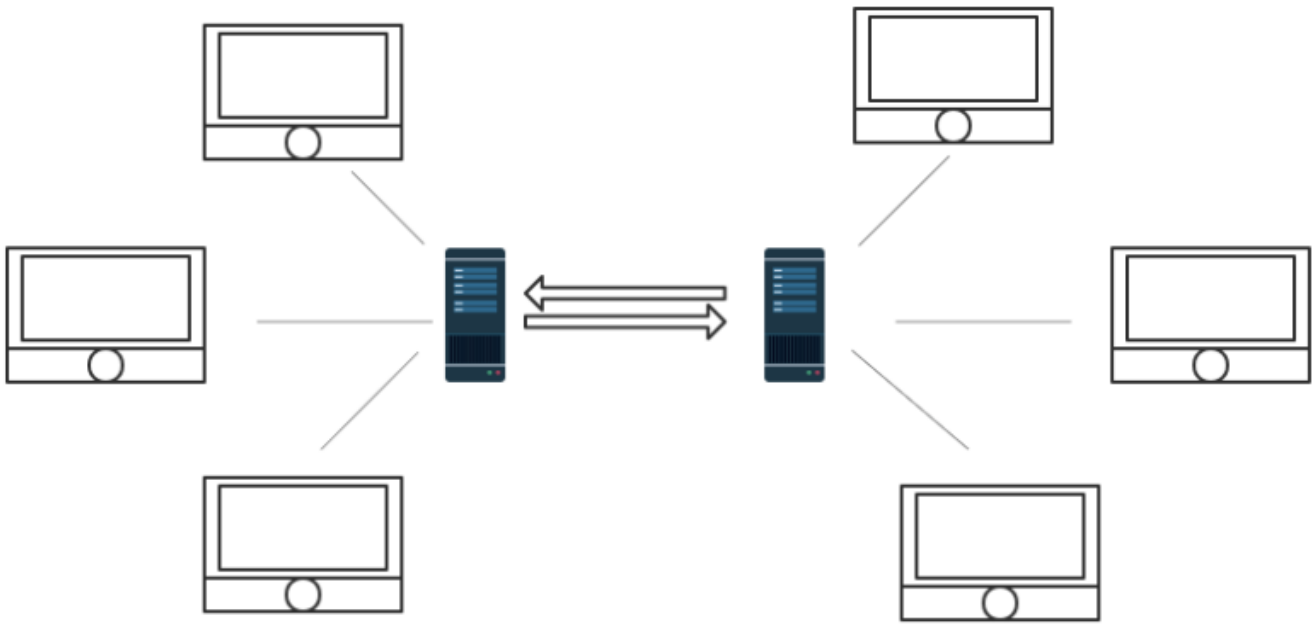
Host主机式 CS

大概在上世纪90年代，在P2P架构的基础上，很自然地诞生了以某个客户端为Host主机（或叫做ListenServer）的CS架构，这样的架构不需要单独都维护一个服务器，任何一个客户端都可以是Sever，能够比较方便的支持局域网内对战，也能节省服务器的运行与开发成本。不过，虽说也是CS架构，如果Host主机不做任何server端的校验逻辑，那么其本质上还是P2P模型，只不过所有的客户端可以把消息统一发送到一个IP，Host再进行转发，这种方式我们称其为Packet Server。



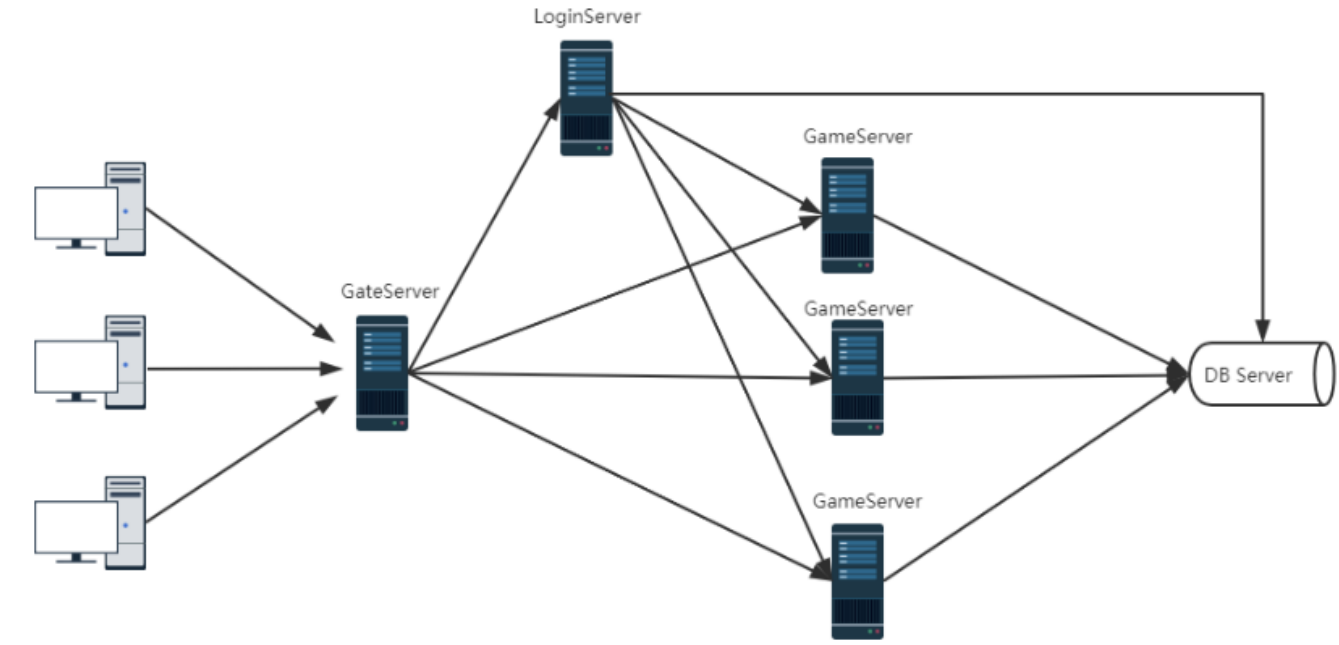
镜像服务器 CS

在2000年后，Eric Cronin的团队在传统多服务器的架构上[2]提出来镜像服务器模型[3]。这种模型提供了多个服务器的拷贝，避免单点崩溃影响到所有玩家的问题。类似CDN，玩家还可以选择就近的服务器进行通信，降低了通信延迟。不过，这种方式增加了服务器的租用和维护成本，在后续的游戏网络架构中并没有被大量使用，倒是WEB服务器广泛采用这种模型并不断将其发扬光大。



分布式 CS

再后来，游戏服务器架构不断发展。游戏存储负载和网络连接负载随后从逻辑服上拆分出来，形成独立的服务；玩家数量增多后，又将游戏拆分成多个平行世界，出现了分服和跨服；游戏逻辑进一步复杂后，又开始按照功能去划分成网关服务器、场景服务器、非场景服务器等。我们今天讨论的网络同步几乎都是在逻辑服务器（基本上无法拆分）上进行的，所以后续的这些架构方式与网络同步的关系并不是很大，这里就不再赘述。



关于帧同步

LockStep，字面意思其实是锁步，基本上是我们所说的帧同步。

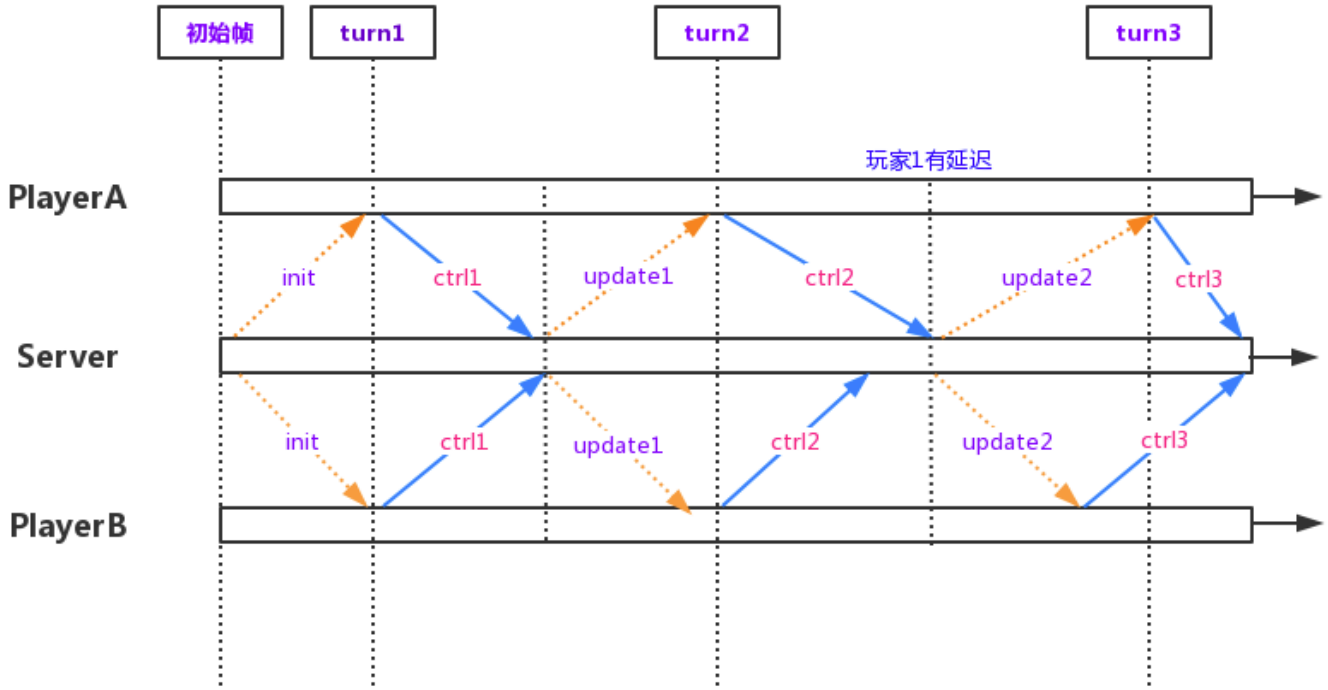
关于 LockStep，有两种主要的LockStep 方式：严格帧锁定、乐观锁。

要做到不同设备的同步结果，还需要满足以下条件：

- 浮点数计算精度问题，需要采取定点数计算；
- 随机数；
- 三角函数的计算，Cos、Sin 等等，需要采用查表法；
- 代码的运行时序严格一致；

严格帧锁定

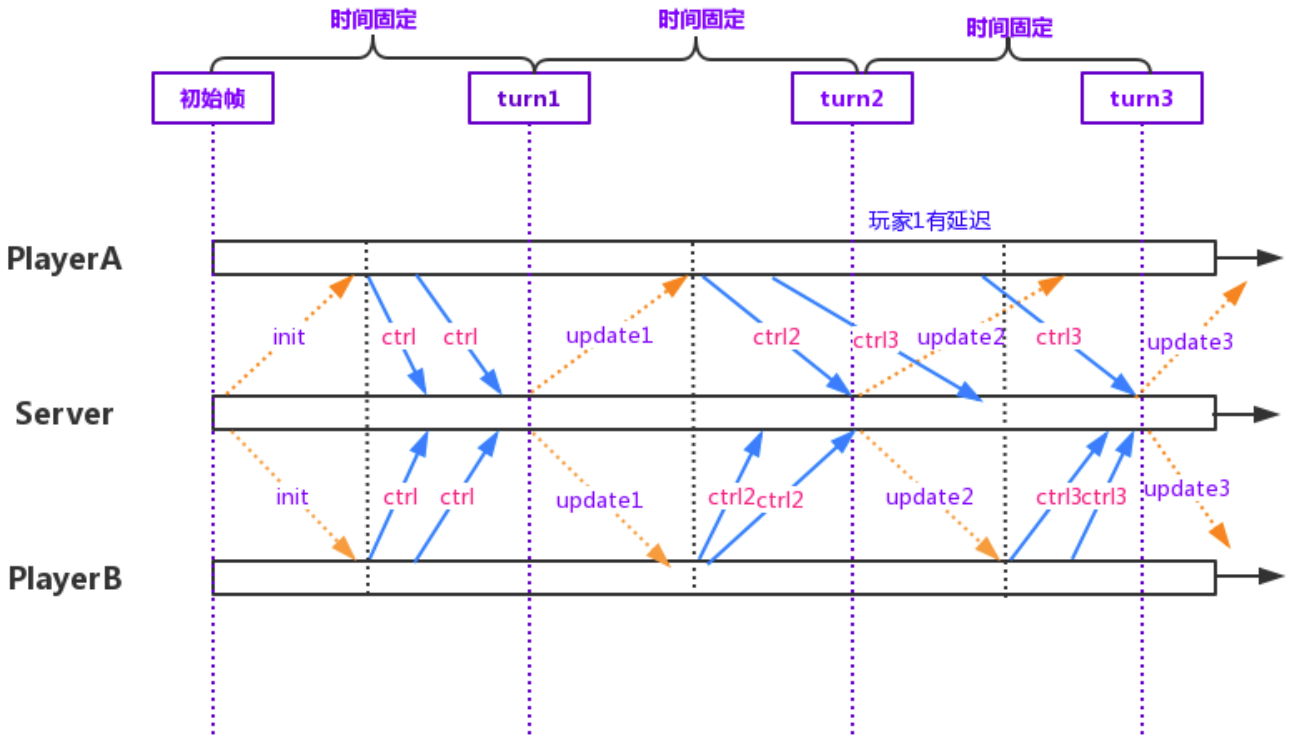
游戏被切成一个个片段，每个片段可以称作 “Turn”，服务器收集到所有玩家的输入后（玩家无输入也可以发送一个空输入），服务器再下发指令，推进客户端逻辑帧的运行。这样的结果是极其严格的，一旦有一个玩家网络有延迟，那么其他玩家必定会等待。



乐观锁

和严格帧锁定一样，游戏也是被切成一个个片段，服务器定时收集客户端的输入，如果有玩家因为延迟而导致输入延迟，那么我们就认为该玩家的输入不在当前 Turn，延迟的操作要么被舍弃，要么被延迟到后面的Turn。这取决于服务器的处理方式。

这是一种乐观的锁定方式，认为玩家的网络波动在一定 Turn 的范围内，是一种可容忍的环境，网络延迟必定带来操作延迟。



浮点数精度误差

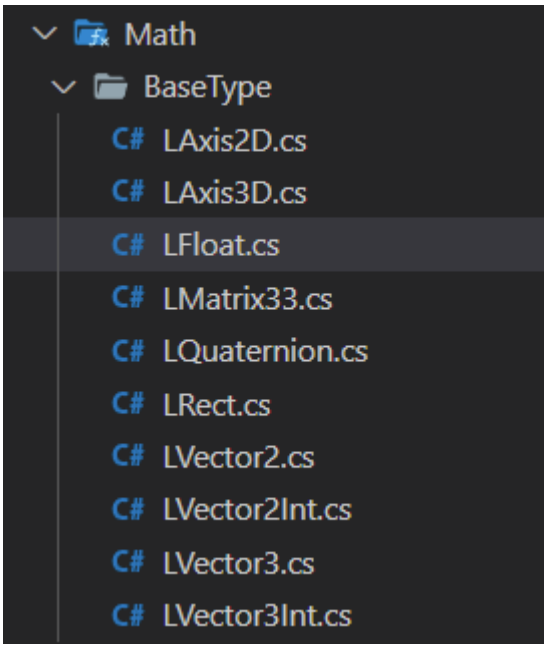
我们游戏中，判断计算的地方，都必须用浮点数转化为放大的整数作为计算结果。

```
1 reference | You, 4 weeks ago | 4 authors (su.tang and others)
public static partial class CLSExtends
{

    64 references
    public static int F2I(this float v)
    {
        return (int)(v * 1000);
    }

    30 references
    public static float I2F(this int v)
    {
        return v * 0.001f;
    }
```

后续也可以完全使用 LFloat，这个是第三方实现的开源定点数数学库。如下图：

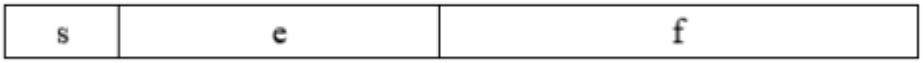


浮点数误差的原因

按照 IEEE754 标准，float 由数符、阶码、尾数组成。

float 总共占 32 位。数符占1位，阶码占8位，尾数占23位。

如下图：



表示公式如下：

$$val = s * 1.f * 2^{e-offset}$$

因此对一些小数的表达会有一些误差。

例如，0.1 可以用 00111101110011001100110011001101 来表示。但是实际转化的值是 0.100000001490116119384765625

float 在10进制内精度只有7位，对应二进制的23位。

```
0.9f = 2^-1+2^-2+....+2^-23;

转化后，2^-1+2^-2+....+2^-23 = 0.89999998;0.89999998
```

double 与 float 对比，如下：

类型	s 位数	e 位数	f 位数	总位数	offset
float	1	8	23	32	127
double	1	11	52	64	1023

另外不同设备 IEEE 可能也存在版本不一样，也可能造成差别。

随机数

这个在我们游戏中，是通过一个特定公司计算，不断更新下一次计算公司的因子，既能得到随机结果，又得到下一次随机的随机种子，从公式的层面上，保证随机序列一样。所以调用时序一致，那么随机就是一致的。

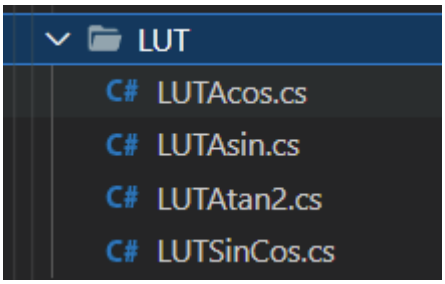
非必要的情况，也可以用上帧号和一下其他天然一致性的数据作为随机数的计算因子，这样既随机，也一致。

三角函数

三角函数和浮点数运算有些不一样，因为使用Unity三角函数库，无法控制过程中的浮点数，如果只对结果进行定点数转化，那么已经来不及了，误差可能已经出现。所以查用查表运算三角函数比较好。

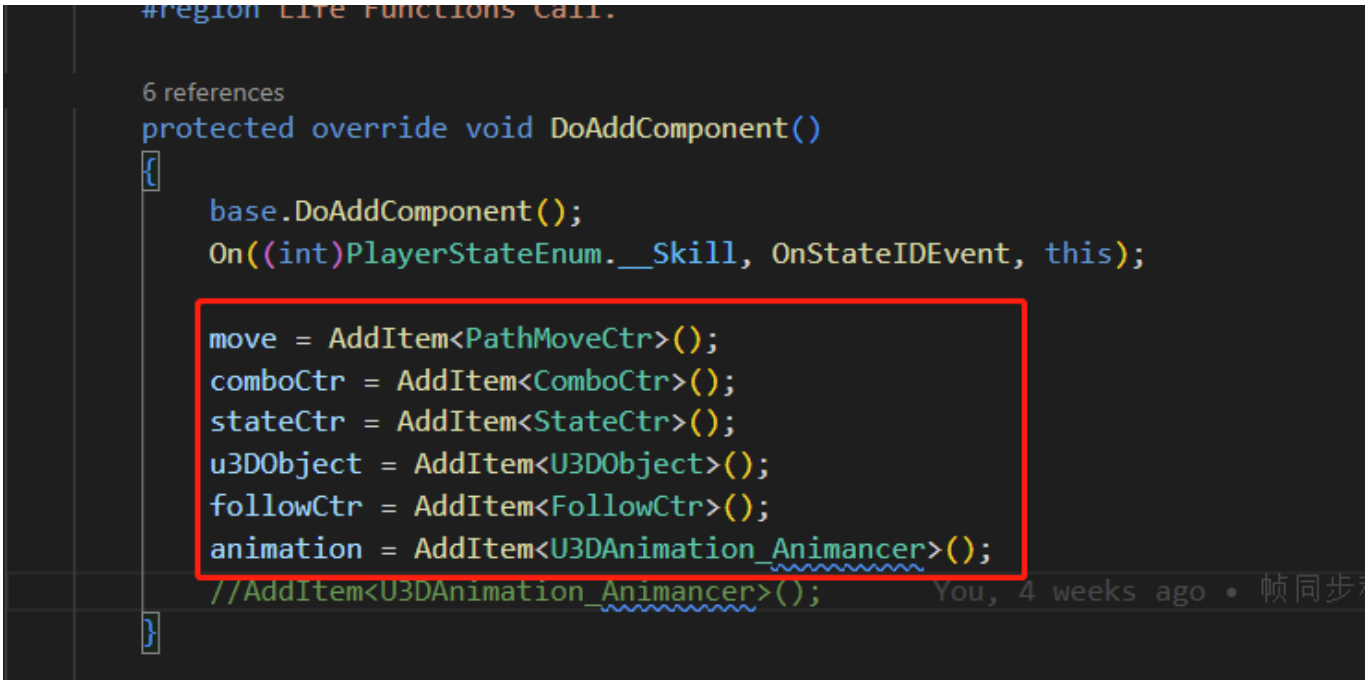
另外查表法，效率也是更高的，这是一种空间换时间的思想。利用一点点的空间存储，即保证结果一样，又节省了算力。

查表三角函数所在，如下图：



代码时序

比较有代表性的是 Component 的顺序，如下图：



```
#region Life Functions Call.

6 references
protected override void DoAddComponent()
{
    base.DoAddComponent();
    On((int)PlayerStateEnum.__Skill, OnStateIDEvent, this);

    move = AddItem<PathMoveCtr>();
    comboCtr = AddItem<ComboCtr>();
    stateCtr = AddItem<StateCtr>();
    u3DObject = AddItem<U3DObject>();
    followCtr = AddItem<FollowCtr>();
    animation = AddItem<U3DAnimation_Animancer>();
    //AddItem<U3DAnimation_Animancer>();      You, 4 weeks ago • 帧同步
}
```

注意，这里的 Component 存在List 中，在逻辑帧进行Update时，也是按照List顺序来的，也就是说，PathMoveCtr先执行，最后 U3dAnimation_Animancer 最后执行。

关于状态同步

状态同步，顾名思义，强调的是状态数据，多数情况下，由服务器下发状态数据，客户端更加状态信息进行效果的表现。

状态同步需要注意以下情况：

- 状态数据的同步压力；
- 客户端的预表现与预测；

状态同步的数据压力

在状态同步下，如果不做特殊处理，那么同步下发的状态数据就是全量包，对于很多MMO游戏来说，这个数据量太大了，所以往往状态同步都是做增量同步以及AOI。

增量同步是指只同步有被修改的数据。

AOI（area of interest），可以理解为只关心小范围数据，例如在一个超大的MMO游戏世界内，我们可以通过目标客户端角色所处的游戏世界位置，定位到一个小范围区域内，也就是说只需要下发小范围区域内的数据。另外再通过白名单和黑名单的方式，将状态数据再筛选一遍，那么同步的数据包又可以减小一部分。

状态同步与帧同步对比

状态同步：

- 网络带宽占用较高，需要转发大量对象的状态，往往都需要同步AOI、增量更新等策略去优化；
- 能做回放，但是数据处理比较麻烦，并且数据量也比较大；
- 服务器多数情况下，需要运行各种判断类的逻辑；

- 不容易出现作弊，可靠性比较高；

帧同步：

- 网络带宽占用较低，往往只转发操作指令；
- 非常适合做回放功能，存储的数据量也比较小，往往只需要存一些输入指令；
- 通常服务器不参与计算，只是纯粹的广播；
- 容易出现作弊，特别是判定逻辑在客户端，例如FPS中的一些锁头挂；

同步问题排查记录

在7月底，开始了一波对历史遗留的PVP问题进行逐个修复 其中主要包括：

- 常规报错，导致运行代码中断;
- 不同设备的浮点数计算误差;
- 回滚过程中，数据的重置清理;
- 行为树的数据使用规范，行为树之间的传值规则;
- 显示层数据不应当作为判定条件;
- 逻辑不严谨，对引用类型的数据做了错误修改;

计算误差

float 浮点数计算、三角函数计算，在不同设备上，无法保证计算的结果一样。后续在做新需求时，也要严格保证计算的一致性结果。

判定来源不可靠

当时存在一个僵直需求的问题，出现这一问题的原因，是逻辑层使用了显示层的数据作为判断，这样很不严谨，如果显示层的数据是在渲染帧进行变化，那么两边设备就不在保证步子一致，最后得出的结果自然就无法保证了。

如下图，数据层的格子坐标一致，显示层的世界坐标不一致：

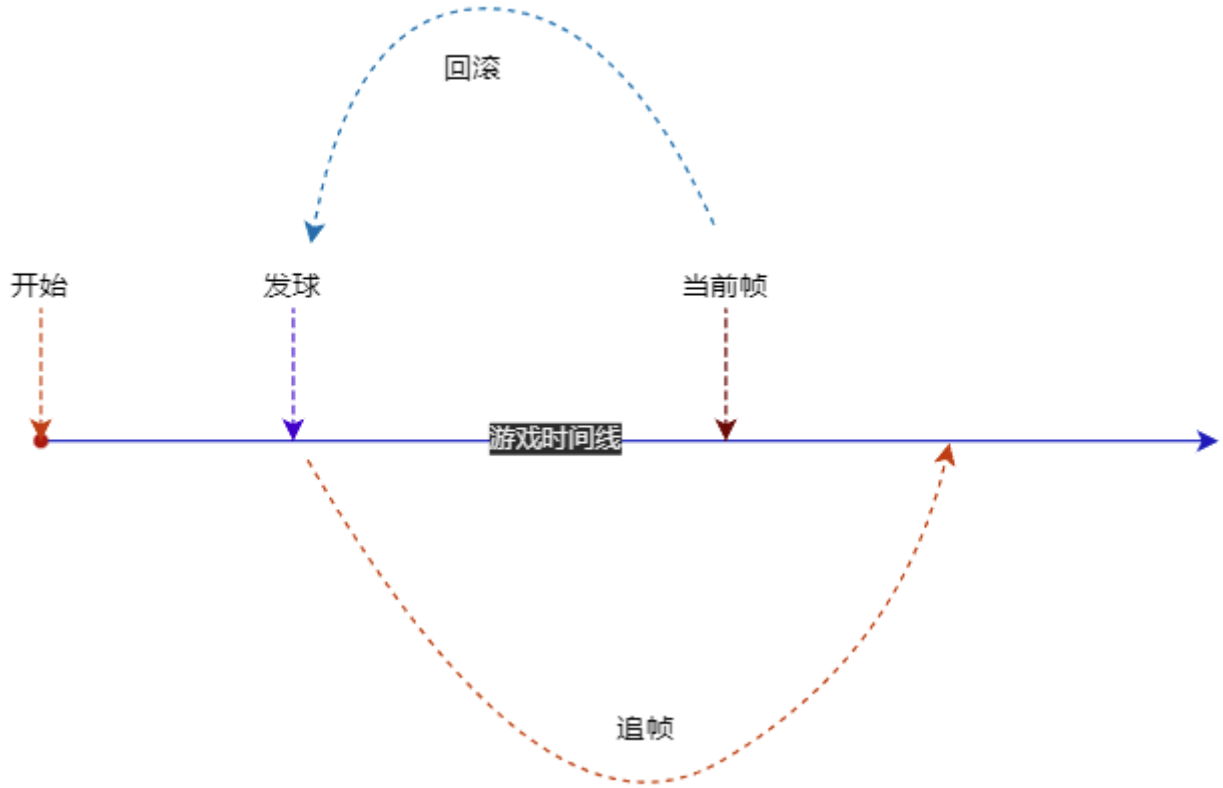
如下图所示，显示层数据是通过 Update 做插值处理，不同设备的帧率无法保证一致，这种波动会导致最终的世界坐标偏差，最终表现为显示层数据误差。

```
Unity Message | 0 references
public void Update()
{
    if (obj._trans && (obj.rts.use || obj._hasChanged))
    {
        if (obj.rts.imm || obj._hasChanged)
        {
            obj.IMMUseRTS();
            obj.rts.imm = false;
        }
        else
        {
            var pos = Vector3.Slerp(obj._trans.position, obj.rts.viewPos, obj.rts.posLerp * Time.deltaTime);
            var rot = Quaternion.Lerp(obj._trans.rotation, obj.rts.viewRot, obj.rts.rotLerp * Time.deltaTime);
            obj.UseRTS(pos, rot);
        }
    }
}
```

游戏流程问题

6、7月的不同步需求内，有大部分不同步原因来自于流程混乱，行为树阶段性的数据没有及时清理，导致数据计算问题的连锁反应。

后续对于回滚点的数据清理要严格控制，行为树的数据传递也需要严格控制。这些都关系到回滚点的定制，我们游戏目前把游戏开局、小节开始、发球三种时间点，作为回滚点进行本地存盘。如下图所示，回滚到发球点。



不严谨的逻辑代码

将引用类型存储起来，下一次取出修改时，只修改了引用类型的内容，但是却没有修改数据的存储位置。

这个问题是之前 PVP 问题中，Point ID 不一致的问题。

这个错误，可以这么解读：

例如，每个人都有 身份证ID 和 姓名等，公安局也会备份这些信息，在二十多年前，公安信息系统没有那么及时和可靠，所以出现冒用身份证、办理假身份证等等事情。

同步问题 Log 排查技巧

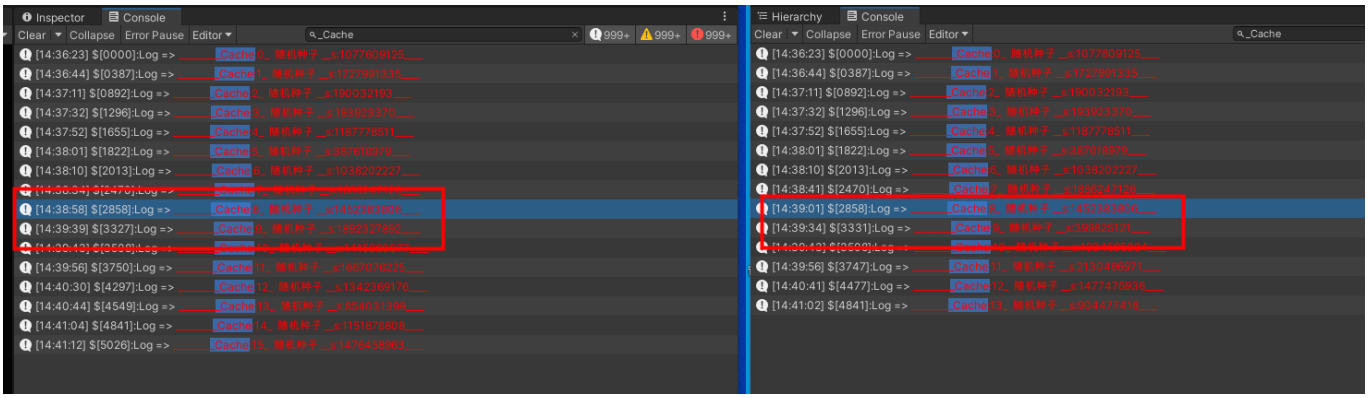
首先我们需要遵守一些规则，如下：

- 只输出关键Log，防止过多无用Log，增加排查难度；
- Log 数据必须带有帧号，帧号是游戏内真正的时间线，是时序判定的唯一途径；
- 必要时 Log 也需要带有角色信息，表明是某个游戏对象输出的Log；

有了以上规则，Log 输出将规范不少。然后我们可以对于两个客户端的日志做搜索。

搜索缓存帧、随机数

搜索 "_Cache"，如下图所示：

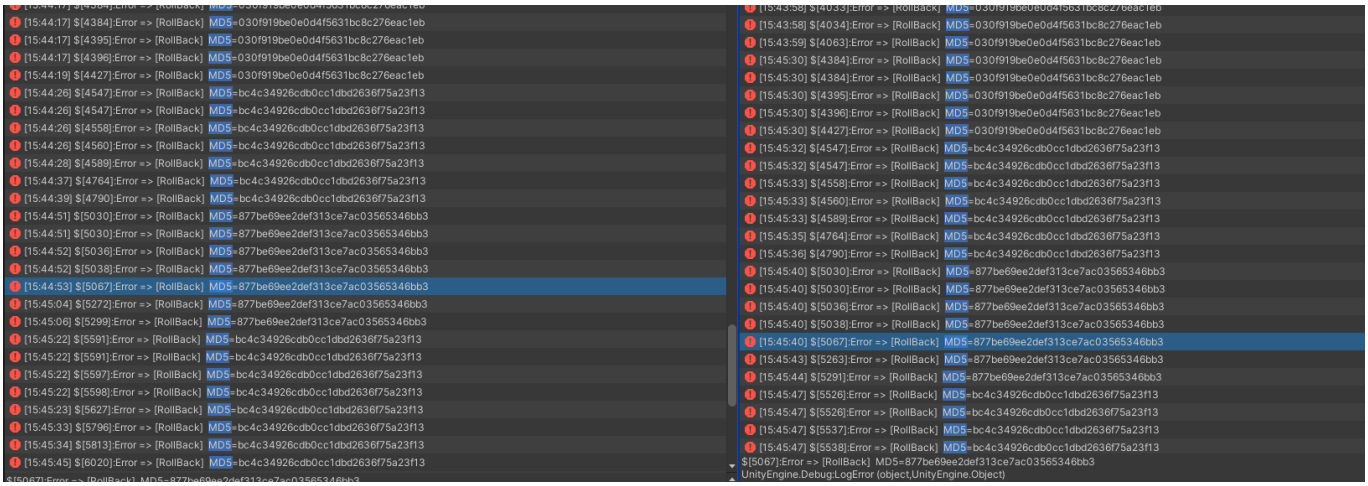


_Cache 是对缓存帧的唯一标识，而缓存帧是回滚数据依据的时间点，通常情况下，每一次回滚都是以最新一次缓存帧作为回滚点，开始回滚时，将游戏内的对象状态数据重置到缓存帧的数据，然后从缓存帧开始，加速追帧到最新的一帧。

如果某一次的缓存帧下的随机数对应不上，说明在这一次的缓存帧之前，出现了一次数据不一致的问题，导致了缓存帧的帧号错位，或者随机数错位。

搜索球员Buff信息

搜索 "MD5="，如下图所示：



MD5 是根据两边队伍的数据，生成的MD5，所以 MD5 必须是帧号与数据的严格一致，才能保证到最终的同步结果。

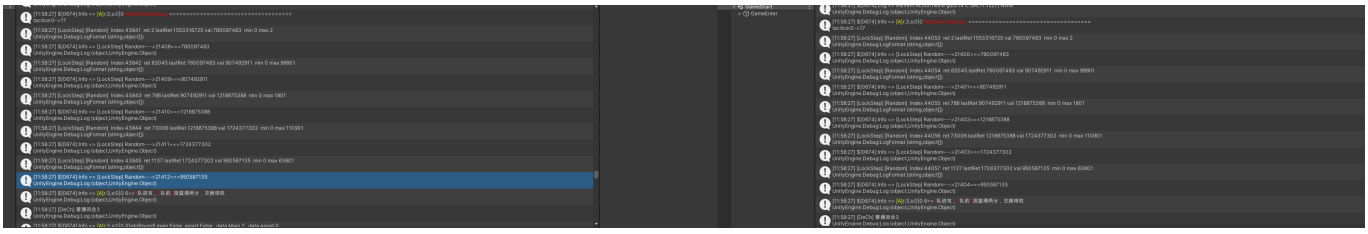
搜索指令信息

搜索 "添加指令" 或 "执行指令"，如下图所示：

指令数据也需要对角色的帧号、所处格子、角色身份验证一致，才能保证最终的数据同步。

随机数时序对齐

搜索 "Random"，如下图所示：



随机数是对一些筛选或假随机的一种特殊实现方式，也必须保证一个帧号与随机数的一致性。

自动化测试工具

有了上文提到的排查Log的方式，那么我们也可以把这些工作交给计算机处理。

大概的设计如下：

- 两个客户端实现自动对局的测试代码，这样可以连续测试多局，无需手动操作；
- 两个客户端各自存储对局Log，带上唯一标识参数；
- 多次对局后，两个客户端把输出的Log进行一对一对比；
- 对比的方式，把帧号当作Key，并且同帧号的数据Log维护成一个队列，保持Log的时序，同步的前提是Log对比一致。如果一致，则本局游戏同步，如果不一致，打出第一行不一致的Log，提供给程序排查，减少人工排查的工作量。

同步异常的定位

多数情况下，测试同学发现不同步或者错误，程序或策划缺少一个快速定位问题的方式，例如回放该局比赛，进行断点，找到异常位置，都是当前比较需要的工具，后期外网的问题更加复杂多变，对于异常问题的定位也需要快速准确。也是后续快发工具的重中之重。