设计风格

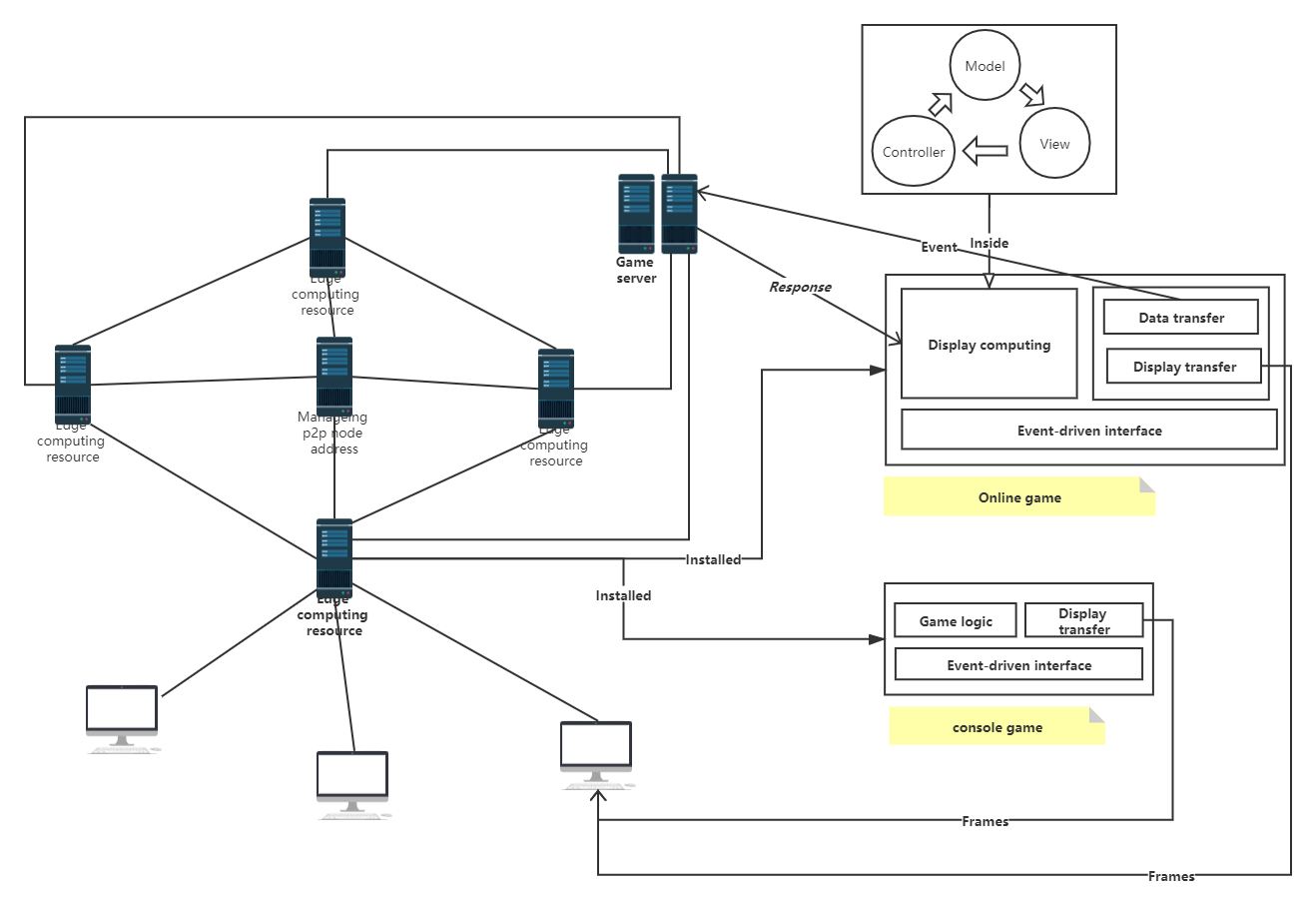
涉及的设计风格：

整体风格：Client-server Style

游戏或视频服务器架构：Layered Architecture

边缘服务器架构：Peer-to-Peer Style

客户端游戏架构：Event Driven Style

其他问题处理：Model-View-Controller (MVC)

1. **整体架构**

参考三层Client-server Style

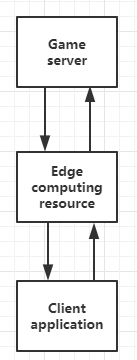
**前提假设：**

因为题目中主要需要讨论的地方是偏客户方面的事务的处理，故对于游戏或视频主服务器的设置，我们只考虑有一台主服务器的情况。

**元件：**

游戏视频服务器 边缘服务器 客户机

**示意图：**



**交互过程：**

客户/服务器模式过程中采取的是主动请求方式：    
  
游戏视频服务器方：

首先服务器方要先启动，并根据请求提供相应服务：    
1. 打开一通信通道并告知本地主机，它愿意在某一公认地址上（周知口，如FTP为21）接收客户请求；    
2. 等待客户请求到达该端口；（这里由于边缘服务器作为客户服务器的中间层，也可将边缘服务器视为客户的代理，即这里的客户指代边缘服务器）    
3. 接收到重复服务请求，处理该请求并发送应答信号。接收到并发服务请求，要激活一新进程来处理这个客户请求（如UNIX系统中用fork、exec）。新进程处理此客户请求，并不需要对其它请求作出应答。服务完成后，关闭此新进程与客户的通信链路，并终止。    
4. 返回第二步，等待另一客户请求。    
5. 关闭服务器    
  
客户方：    
1. 打开一通信通道，并连接到服务器所在主机的特定端口；（同上，这里边缘服务器作为游戏服务器的代理，客户端认为边缘服务器即为链接的目标服务器）  
2. 向服务器发服务请求报文，等待并接收应答；继续提出请求......    
3. 请求结束后关闭通信通道并终止。

边缘服务器：

作为中间层，指代上述两方分别指的客户和服务器。

在处理请求时，会进行数据处理操作。如：收到客户端的数据后，会将庞大的数据计算或筛选，处理成游戏服务器可更方便处理的结果，再将其发送到游戏服务器。收到游戏服务器的数据后，也经过处理只向客户端发送必要的数据。这其中，边缘服务器也发挥着控制游戏的作用，换句话说，它像是一个小型的拥有部分逻辑处理功能的游戏服务器，这一点我们后面会详细地介绍。

**设计缘由**：

由于计算资源瓶颈，我们不把大量的计算或数据处理放在客户端，又因为时延的问题，即服务器和客户端之间数据交互所需时间可能过长，或因其他问题导致延迟，给客户造成不好的游戏体验，我们在客户与服务器之间设计了边缘服务器作为中间层，也负责处理一部分的逻辑，让客户端的游戏反映更为迅速，同时也减轻了服务器端的负担，整体的网络负载也极大的减少。同时，因为传输数据量的减少，可以更保证数据的安全性。此外，当游戏服务器宕机时，边缘服务器可暂存部分数据，提高了整体的鲁棒性。

1. **游戏视频服务器架构：**

参考Layered Architecture

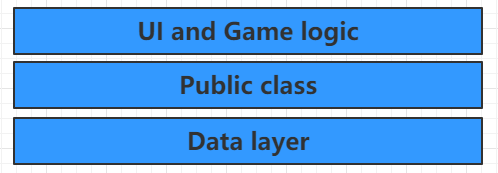
**前提假设：**

首先视频服务器较为简单，可以包括在游戏服务器里，故我们只详细说明游戏服务器。这里我们对游戏类型（网游，单机游戏）做一般化处理，都视作，游戏服务器只对关键数据进行存储或处理，如玩家的基本信息，胜负情况，以往战局信息，好友，永久性装备等，即在游戏操作中非关键的数据，以及主要的游戏实体做数据存储或处理，如玩家之间的攻击判定等。而对于边缘服务器，它们相当于常见的电脑主机，可以同时运行计算多项应用。除了计算功能，它们还有存储备份功能，比如类似绝地求生或守望先锋的对局式游戏，我们把一局内的某些数据放在边缘服务器上；或比如开放性地图的游戏，可把玩家所在某一范围内区域的数据放在边缘服务器上，这一点我们将在后面详解。

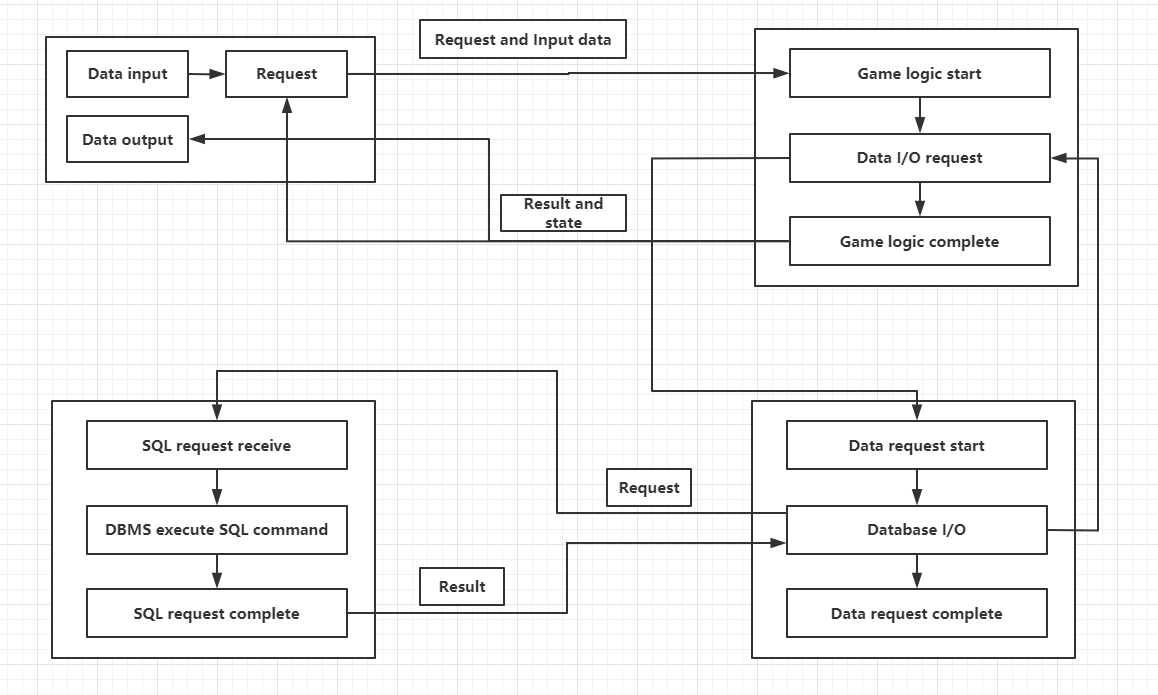
**元件：**

公共类库层 数据层 UI以及业务逻辑层

**示意图：**



表示层负责处理用户的输入和向客户的输出（出于效率的考虑，它可能在向上传输用户的输入前进行合法性验证）。业务逻辑负责建立数据库的连接，根据用户的请求生成访问数据库的 SQL 语句，并把结果返回给客户端。数据层负责实际的数据库存储和检索，响应功能层的数据处理请求，并将结果返回给功能层。



**交互过程：**

游戏服务器偏重于公共类库层和数据层，而边缘服务器偏重于UI及业务逻辑层。对于计算方面，边缘服务器就相当于代替客户端运行着游戏应用，并将计算得到的画面转化为视频流，输送给客户端进而呈现。关于数据方面，因为边缘服务器上的数据主要作用是支撑玩家完成当前的游戏，而非在永久的数据存储，或支持日后的游戏继续，所以这里的数据存储只是暂时性的，并且在完成一局游戏，或一阶段时间后的游戏，或者进入到了又一个不同的游戏区域后，会向游戏服务器传输玩家的某些必要数据，如打败多少敌人，玩家保留了获得的哪些装备等。此外，它还负责处理客户端的画面显示即UI的逻辑处理。与之相反，游戏服务器可间断性地向边缘服务器更新一些玩家请求的或必要的数据，而不必过多地注重UI。

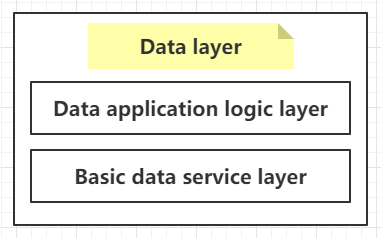
1.公共类库层

公共类库包括一些通过类库和一些跟现有系统相关的引入组件（不依赖于当前系统）

把公共类库作为最底层，是为了实现其他高级层对其引用，由于公共类库层不依赖与现有系统，故而可以被所有模块引用

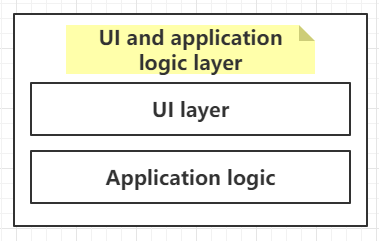
2.数据层

数据层处理是系统的数据处理中枢，数据层可以细分为两个小层：基础数据服务层、数据业务逻辑层



基础数据服务提供类似于数据的存储（数据库或文件存储）的公共基础服务（如类似与三层的Model层和DAL层，提供通过数据访问接口，以及数据对应实体），而数据业务逻辑层则为更高层（UI以及业务逻辑层）提供数据服务接口（类似于BLL层，为更高层组织数据形式）。

3.UI以及业务逻辑层



UI业务逻辑层可细分为：业务逻辑层和UI层，业务逻辑层主要处理UI层的交互逻辑以及对应的业务逻辑，UI层和业务逻辑层可以算是处于同一层

**设计缘由**：

由于在这种形态中三层是分别放在各自不同的硬件系统上的，所以灵活性很高，能够适应客户机数目的增加和处理负荷的变动。例如，在追加新业务处理时，可以相应增加装载功能层的服务器。因此，系统规模越大这种形态的优点就越显著。 这对于玩家量大的游戏或者并发性高的游戏来说，是非常适合的。

1. **边缘服务器架构**

参考Peer-to-Peer Style，中心化拓扑（Centralized Topology）；

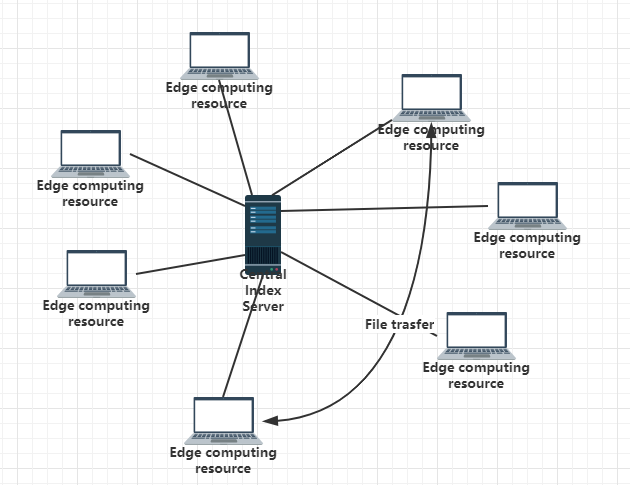
**前提假设：**

多个边缘服务器形成p2p网络，某地理区域内，由一个中心服务器所管控。中心服务器有所有所连边缘服务器的目录，并会定期检查更新边缘服务器的状态。特殊情况时，还可以对边缘服务器进行调度。边缘服务器上，分为90%的计算存储资源，以及10%的存储资源。

**元件：**

边缘服务器 Central index server

**示意图：**



**交互过程：**

*边缘服务器：*

计算存储资源主要负责处理与客户端的UI交互的数据，是将游戏画面转化成视频流输送到客户端，负责保证用户端画面的流畅，逻辑的判断正确。计算存储资源和存储资源都会存储当局游戏，或当前范围内游戏的数据。以确保当此服务器宕机时，玩家当前状态的数据不会丢失。甚至最坏的情况，即使某资源上当前状态丢失，也会有另一资源保有备份，不会全部丢失。

边缘服务器与客户端之间存在着延迟检测，当遇到客户端出现延迟时，边缘服务器会迅速报告给中心目录服务器，若客户端延迟在一个极小的时间区段内没有恢复，则中心服务器查找邻近可以服务器，并将信息传输给当前边缘服务器，此边缘服务器即可由存储资源部分将该客户数据迁移给目标服务器，完成服务器客户数据迁移。除了延迟检测，还有信号强度检测，这一检测可以解决客户的移动问题，当客户信号强度较弱达到某个数值时，与上述方法一致，会迅速报告给中心目录服务器，进而迁移数据并转移客户至新的边缘服务器。

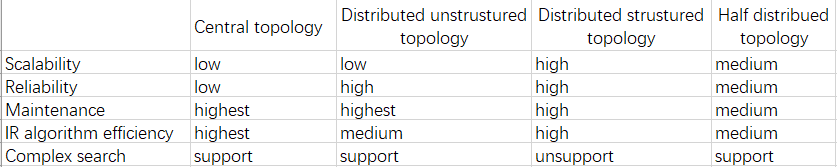
当客户所在地区边缘服务器负荷状态较满时，若有新的用户想加入服务器，则首先判断该用户是想观看VR视频还是玩VR游戏，若是观看视频，则可通知中心目录服务器，将其直接连入最近的可用边缘服务器。若是玩VR游戏，则判断是否存在观看VR视频的用户，因为VR视频的观看所需求的即时性不高，所以我们可以先对观看视频的用户做一些妥协。如在新用户加入之前的计算处理中，预处理视频资源，并将其放在存储资源上，并通知中心目录服务器找好最近可用的边缘服务器。这样新加入的用户可以替换掉视频计算所占的资源，而预存储的视频通过调取存储资源传输给用户。当看视频的用户有观看新视频的请求时，则直接在新的边缘服务器上进行运作，既保证满足了用户看视频的需求，也保证了玩VR游戏的玩家链接到的是速度最快的服务器。

*Central index server：*

中央目录服务器，为网络中各节目提供目录查询服务，传输内容无需再经过中央服务器。这种网络，结构比较简单，中央服务器只负责去安排调度或搜寻边缘服务器，所以负担大大降低。由于目录集中管理，对于网络的管理和控制上是一种可选择方案。比如在应对某一边缘服务器宕机时，中心目录服务器可以快速地找到离玩家最近的另一边缘服务器，通知玩家与新的边缘服务器，直接将玩家链接到其上，并将在之前边缘服务器上的数据迁移到新的边缘服务器上。这里的数据迁移即为上述提到的过程。

**设计缘由**：

就p2p结构来说，健壮性，具有耐攻击、高容错的优点。由于服务是分散在各个结点之间进行的，部分结点或网络遭到破坏对其它部分的影响很小。P2P网络一般在部分结点失效时能够自动调整整体拓扑，保持其它结点的连通性。隐私保护，由于信息的传输分散在各节点之间进行而无需经过某个集中环节，用户的隐私信息被窃听和泄漏的可能性大大缩小。负载均衡，减少了对传统C/S结构服务器计算能力、存储能力的要求，同时因为资源分布在多个节点，更好的实现了整个网络的负载均衡。就中心化拓扑结构来说，维护简单，资源发现效率高。由于资源的发现依赖中心化的目录系统，发现算法灵活高效并能够实现复杂查询。中心化拓扑虽然相比于其他结构，可扩展性低，可靠性也不高，但是这两项都不是它主要的业务，它的主要业务就是要做到维护性好，可以长久地发挥作用并储存数据，以及发现算法效率高，能够在边缘服务器故障时，迅速发现下一个可用节点即可用边缘服务器。



1. **客户端游戏架构**

参考Event Driven Style

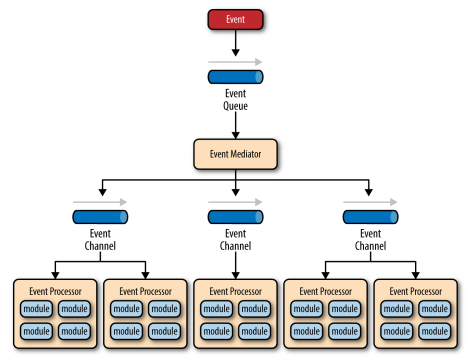
**前提假设：**

同理，视频方面视为包括在游戏里，这里只对游戏进行说明。客户端游戏主要负责UI展示，以及对接收的数据进行处理和对接相应的画面元素，进而产生响应，同时也会根据用户的操作更改数据处理画面，并将必要的数据传输给边缘服务器。通俗地讲，VR眼镜相当于平时的显示器，操作器即为鼠标与键盘，需要传输的也正是这些操作器产生的信号。

**元件：**

操作器 客户机 VR眼镜

**示意图：**



**交互过程：**

## *输入方式：*

## VR输入设备颠覆式的概念是空间含义，它区别于传统输入设备最直接的意义是将输入参数空间化，即允许开发者跟踪用户的手柄空间位置、轨迹等创造出更多的可能性。

Menu Button：

菜单按键是一个统一化功能性按键，根据HTC VIVE对其的命名来看，在对其按键功能的确定上，不应该存在更多偏离于菜单概念的定义存在，即菜单按键只是用于呼出菜单。而有趣的是在VR中控制器是俩个，分为左右手。这样就允许我们分化左右手菜单键用于不同界面的呼出，可以更好的简化界面层级深度。

Trackpad：

Trackpad，顾名思义，它是一个触摸板。提供了四个方向概念的按键操作（上下左右），也提供了更加细腻的面板上滑动坐标轨迹操作。这样我们可以把其当作上下左右模式化的位置移动工具，也可以把其当作细微坐标的细腻调整工具使用。

Trigger：

扳机键提供了一个0-1之间的动态数值，产生于体验者扣动扳机的整个过程。它也是游戏中使用最为频繁的按键，主要功能用于确认操作。

Grip Button：

侧面按键，从位置上来看应该当作不常用的辅助按键功能使用。用于一些必须但并不是常用的游戏行为控制。

控制器的空间六自由度：

控制器的六自由度是以上几种输入模式中唯一一个真正的VR概念输入，因为它是三维化的。简单来说六自由度为开发者提供感知控制器的旋转角度和虚拟空间的坐标，这样就提供于游戏更加三维化概念的输入模式。六自由度的运用更大程度上会与以上提到的操作结合使用。

## *输出方式：*

力反馈输出：

在HTC VIVE中控制器提供了震动功能，它可以提供我们用于玩家游戏交互中的力反馈。手柄的震动并不是第一次在游戏的输出方式中出现。主机游戏机的手柄以及手机都存在震动功能，而震动功能在游戏中的应用也并不是第一天出现。

图像显示输出：

HMD（头戴显示设备），它提供了全景观看虚拟现实的概念，即屏幕显示三维化。它包含了头部空间跟踪定位，以及封闭式、更大视场的体验。这区别于传统游戏设计（分镜设计），因为自由度的关系场景中将存在无数个分镜，在设计上天然的要求了产品无视觉死角。

音频输出方式：

音频输出在VR中是一个极其需要注意的事项，它也是一种三维化状态的需求。音频播放点与玩家头部模型的位置以及环境状态，决定了音频传递到用户耳中的感受。

**设计缘由**：

支持软件重用，容易实现并发处理。具有良好的可扩展性，通过注册可引入新的构件，而不影响现有构件。可以简化客户代码，简化客户的设备需求，低配置依旧可以体验到需要高计算才能达到的效果。整体来说适合VR游戏的运作，也适合VR视频的观看。

1. **其他问题处理**

Model-View-Controller (MVC)

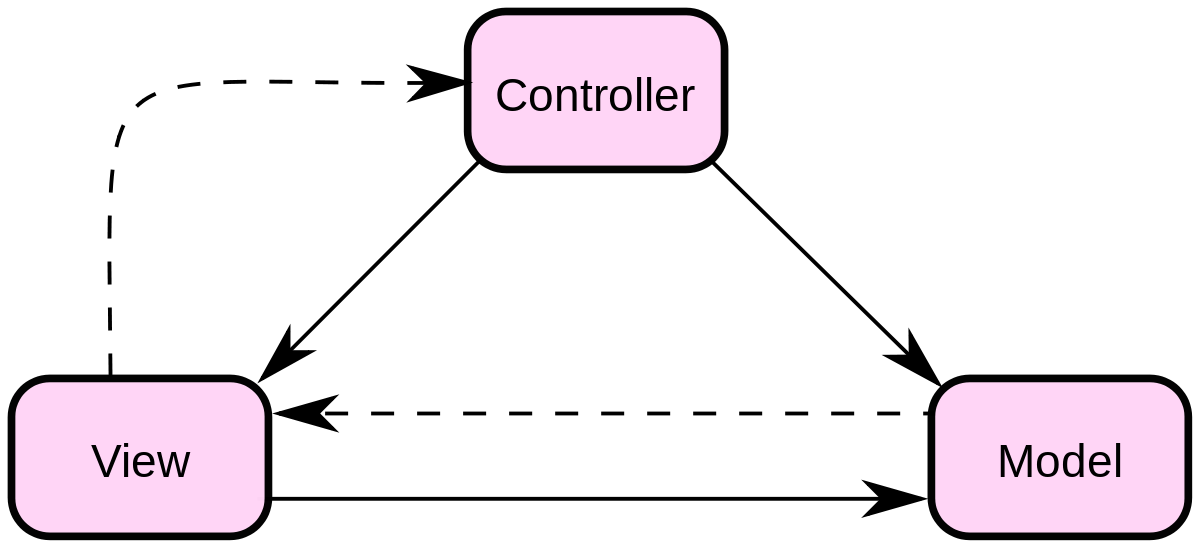
**前提假设：**

客户端作为view层，边缘服务器作为controller层，游戏视频服务器作为model层。

**元件：**

边缘服务器 客户端 游戏视频服务器

**示意图：**



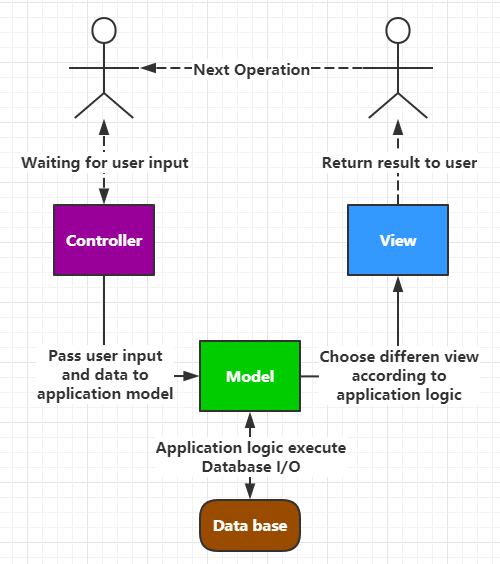
Model（模型） - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。

View（视图） - 视图代表模型包含的数据的可视化。

Controller（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

**交互过程：**

客户端作为view层，边缘服务器作为controller层，游戏视频服务器作为model层。



**设计缘由**：

在正常游戏的过程中，客户端的画面不会出现明显的卡顿，保证画面的流畅运行。当游戏服务器出现故障时，边缘服务器可以继续维护着画面的运行，即视频流的传输。为了确保客户端的画面不会停止，或干扰正常游戏的进行，这个模式可以在model层出现问题时，view和controller层依旧保留部分功能运行，让玩家得到良好的体验。[耦合性](http://baike.baidu.com/item/%E8%80%A6%E5%90%88%E6%80%A7)低，视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型和控制器代码，同样，一个应用的业务流程或者业务规则的改变只需要改动MVC的模型层即可。重用性高：[MVC模式](http://baike.baidu.com/item/MVC%E6%A8%A1%E5%BC%8F)允许使用各种不同样式的视图来访问同一个服务器端的代码，因为多个视图能共享一个模型。部署快，[生命周期](http://baike.baidu.com/item/%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F)成本低，MVC使开发和维护用户[接口](http://baike.baidu.com/item/%E6%8E%A5%E5%8F%A3)的技术含量降低。使用MVC模式使开发时间得到相当大的缩减，