阶段一：学习使用框架

阶段二：使用框架实现游戏业务

阶段三：写框架

阶段四：商业化部署

学习使用框架的方法：

- 读文档
- 装环境
- 写用例

zinx描述

zinx框架是一个处理多路IO的框架。在这个框架中提供了若干抽象类，分别在IO处理的多个阶段生效。开发者可以重写抽象类中的虚函数完成自己需求的处理功能。

zinx框架的使用步骤

1. ZinxKernel::ZinxKernelInit() 初始化框架
2. 写类继承AZinxHandler，重写虚函数，在函数中对参数进行处理（比如将参数内容打印到标准输出）
3. 写类继承Ichannel，重写虚函数完成数据收发，重写GetInputNextStage 函数，返回第二步创建类的对象
4. 添加步骤3类创建的对象到框架中
5. 运行框架

标准输入回显标准输出的编写思路

1. 创建三个类：标准输入类，回显类，标准输出类
2. 重写标准输入类的读取函数
3. 重写回显类处理函数
4. 重写标准输出类的写出函数
5. 创建以上三个类的全局对象（堆对象），添加通道对象到框架（kernel）
6. 运行框架

添加命令处理类

1. 创建命令处理类继承AzinxHandler，重写处理函数和获取下一个处理环节的函数
2. 处理函数内，根据输入内容不同，要么添加输出通道，要么摘除输出通道
3. 获取下一个处理环节函数中，指定下一个环节是退出或回显
4. 设定输入通道的下一个环节是该类对象

添加日期前缀

1. 创建添加日期类，继承AzinxHandler。重写处理函数和获取下一环节函数
2. 处理函数：将日期和输入字符串拼接后，new一个对象返回
3. 获取下一环节函数：返回回显对象
4. 在命令处理类的处理函数中：根据输入命令设置当前是否要添加前缀的状态位
5. 在命令处理类的获取下一环节函数中，判断当前状态，需要添加前缀--》返回添加日期前缀的对象；不需要添加前缀--》返回回显对象

需要调用的框架静态函数

- 初始化，去初始化 `ZinxKernel::ZinxKernelInit()` 和 `ZinxKernel::ZinxKernelFini()`
- 运行框架 `ZinxKernel::Zinx_Run()`
- 通道添加和摘除 `ZinxKernel::Zinx_Add_Channel()` 和 `ZinxKernel::Zinx_Del_Channel()`
- 退出框架 `ZinxKernel::Zinx_Exit()`

多个AzinxHandler对象之间的信息传递

- 数据封装成IzinxMsg类在多个AzinxHandler对象之间传递
- 使用时，要现将IZinxMsg类型引用动态转换成所需类型引用

zinx框架处理数据的本质

- 将数据在多个AzinxHandler对象之间传递，挨个处理
- 传递的规则通过重写GetNextHandler函数定义

三层结构重构原有功能

1. 自定义消息类，继承UserData，添加一个成员变量szUserData
2. 定义多个Role类继承Irole，重写ProcMsg函数，进行不同处理

3. 定义protocol类，继承Iprotocol，重写四个函数，两个函数时原始数据和用户数据之间的转换；另两个用来找消息处理对象和消息发送对象。
4. 定义channel类，继承Ichannel，在getnextinputstage函数中返回协议对象

添加关闭输出功能

1. 写一个关闭输出的角色类，摘除输出通道或添加输出通道
2. 在CmdMsg用户数据类中添加开关标志，是否是命令标志
3. 在协议类中，根据输入字符串，设置开关标志和是否是命令的标志
4. 在协议类分发消息时，判断是否是命令，是命令则发给关闭输出角色类，否则发给回显角色类

添加日期前缀管理功能

1. 写日期管理类，处理命令时，改变当前状态。处理非命令时，添加日期前缀后不添加日期前缀后，将数据传递给下一环节（echo对象）
2. 初始化日期管理类时，设置echo对象为下一个环节
3. 修改命令识别类，命令消息传递给输出通道控制类，非命令消息传递给日期前缀管理类
4. 设定输出通道控制类的下一个环节是日期前缀管理类

添加TCP方式的数据通信

1. 创建tcp数据通道类继承ZinxTcpData，重写GetInputNextStage 函数，返回协议对象
2. 创建tcp连接工厂类继承IZinxTcpConnFact，重写CreateTcpDataChannel ，构造步骤1的对象
3. 创建ZinxTCPListen类的对象，指定端口号和工厂对象（步骤2定义的类的对象），添加到kernel中

- Ichannel对象读取到的数据给谁了？
- 给该对象调用GetInputNextStage函数返回的对象
- Iprotocol对象转换出的用户请求给谁了？
- 给该对象调用GetMsgProcessor函数返回的对象

timerfd产生超时事件

- timerfd_create()返回定时器文件描述符
- timerfd_settime()设置定时周期，立刻开始计时
- read，读取当当前定时器超时的次数，没超时会阻塞，
- 一般地，会将定时器文件描述符结合IO多路复用使用

定时器管理类

- 处理超时事件：遍历所有定时任务，计数减一，若计数为0，则执行该任务的超时处理函数
- 添加定时任务
- 删除定时任务
- 作为timer_channel的下一个环节

时间轮定时器

- vector存储轮的齿
- 每个齿里用list存每个定时任务
- 每个定时任务需要记录剩余圈数
- 时间轮类中要有一个刻度，每秒进一步

时间轮添加任务

- 计算当前任务在哪个齿上
- 添加该任务到该齿对应的list里
- 计算所需圈数记录到任务中

时间轮删除任务

- 遍历所有齿
- 在每个齿中遍历所有节点
- 若找到则删除并返回

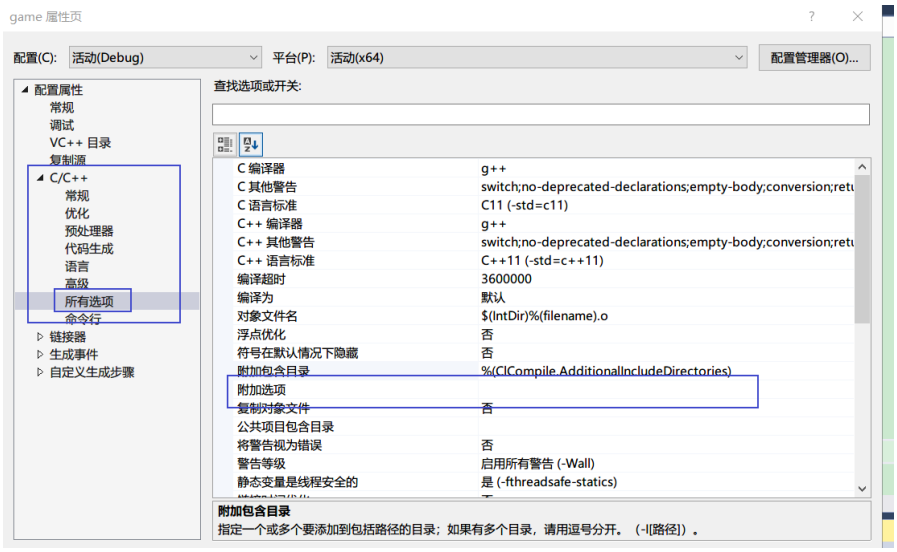
时间轮调度处理

- 移动当前刻度
- 遍历当前齿中的任务列表
    - 若圈数为0，则执行处理函数，摘除本节点，重新添加
    - 否则，圈数--

游戏服务分层

- 通道层创建和维护游戏客户端的TCP连接
- 协议层，接收字节流，产生游戏相关的请求；将需要客户端处理的游戏请求转换成字节流
- 业务层：根据接收消息不同，进行不同处理(角色类的对象和通道对象绑定)角色类对象存储对应玩家的数据
- 消息定义：继承userdata类之后，添加一个成员存储当前的游戏消息（google::protobuf::message）

编译protobuf



TCP粘包处理

1. 数据要有边界
2. 缓存，将未处理报文缓存，将新报文续到缓存报文尾部
3. 按照报文要求，一边处理一边滑窗

游戏服务总体架构：