

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

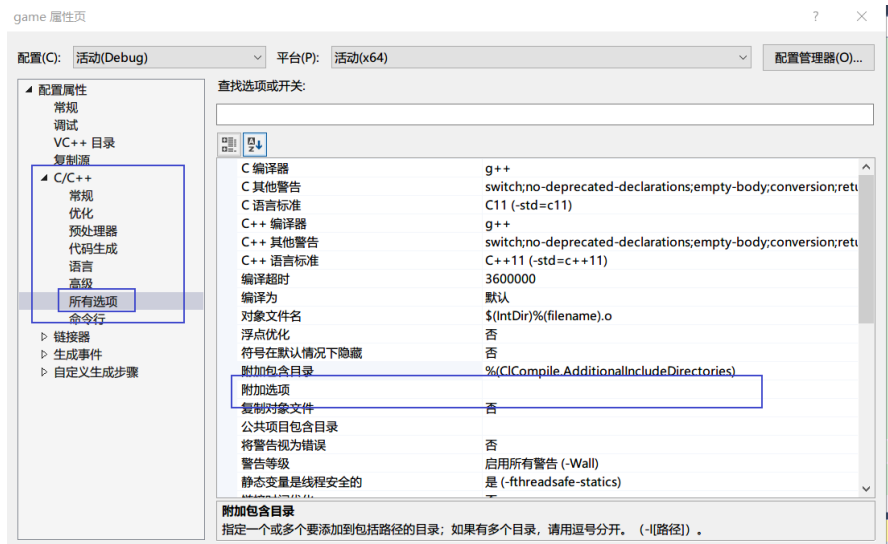
## 时间轮调度处理

- 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)

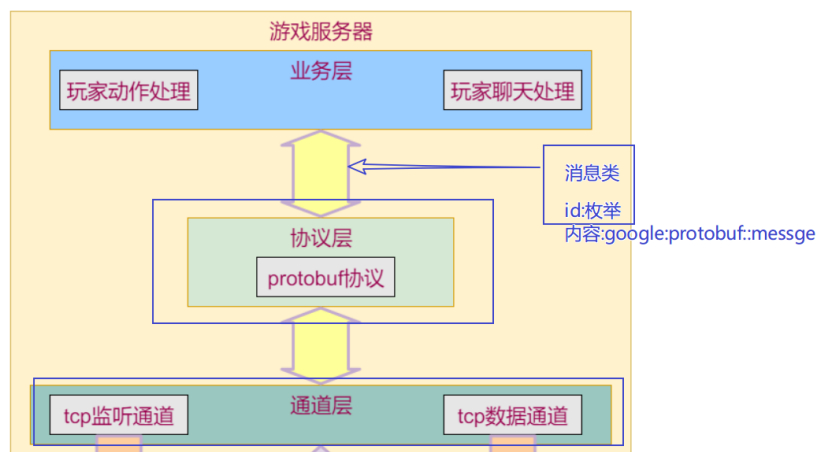
## 编译protobuf



## 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：



## 网格法AOI

- 目的：获取周围玩家
- 模型：将游戏世界的坐标分割成网格，玩家属于某个网格
- 周围：玩家所属网格周围8个相邻网格内的玩家
- 游戏世界矩形：包含固定数量网格对象的容器
- 网格对象：包含若干玩家的容器
- 玩家：拥有横纵坐标的对象

## 游戏世界类实现

- 构造函数：边界相关属性的赋值，创建格子们
- 添加玩家的函数：计算玩家所属格子，push\_back
- 删除玩家的函数：计算玩家所属格子，remove

|           |         |           |
|-----------|---------|-----------|
| n-1-x轴网格数 | n-x轴网格数 | n+1-x轴网格数 |
| n-1       | n       | n+1       |
| n-1+x轴网格数 | n+x轴网格数 | n+1+x轴网格数 |

## AOI结合GameRole类

- 继承player类，重写getx和gety---》返回z坐标
- 创建唯一游戏世界对象（全局对象）
- gamerole初始化（init函数）时添加自己到游戏世界
- 去初始化时，摘除自己

## 设置protobuf类型消息的repeated类型

- add\_XXXX函数
- 调用后，会向当前消息添加一个数组成员，返回数组成员的指针

## 设置protobuf中复合类型

- mutable\_xxxx函数
- 调用后，会向当前消息添加子消息。返回子消息的指针

## 连接到来（玩家初始化）时

- 属性pid赋值为socket值
- 属性name写成tom
- 初始坐标100,100
- 向自己发内容是ID和姓名的1号消息
- 向自己发内容是若干周围玩家信息的202号消息
- 向周围玩家发送内容是自己位置的200号消息

## 世界聊天思路

游戏相关的核心消息处理逻辑都是要在该类中实现的。

需求回首：

为

- 新客户连接后，向其发送ID和名称
- 新客户连接后，向其发送周围玩家的位置
- 新客户连接后，向周围玩家发送其位置
- 收到客户端的移动信息后，向周围玩家发送其新位置
- 收到客户端的移动信息后，向其发送周围新玩家位置
- 收到客户端的聊天信息后，向所有玩家发送聊天内容
- 客户端断开时，向周围玩家发送其断开的消息

GameRole::ProcMsg

ZinxKernel::GetAllRole

关键字：周围。

以上所列出的需求，基本都是这样的套路：在XXX的时候，发送XXX给XXX。 zinxkernel::sendout()

- 发送时机
- 消息内容
- 发送对象：怎样表示周围玩家？

## 5.1AOI设计与实现

## 玩家移动处理

- 广播新位置给周围玩家
- 若跨网格，视野切换（获取移动前周围玩家S1，获取移动后的周围玩家S2）
  - 新邻居：互相能看见（{x|x 属于S2 && x 不属于S1} --> 发送200号消息
  - 旧邻居：互相看不见（{x|x属于S1 && x不属于S2} --> 201号消息

## C++随机数

- default\_random\_engine, 构造时传入种子
- () 重载, 返回随机数

## 用valgrind查内存泄漏

- valgrind --leak-check=full --show-leak-kinds=all 程序
- 等待程序退出后显示内存报表
- 关注报表中必然泄漏的那一项

- ```
definitely lost: 0 bytes in 0 blocks
indirectly lost: 0 bytes in 0 blocks
possibly lost: 0 bytes in 0 blocks
```
- 查看调用栈确定哪个函数泄漏了

## 玩家全部退出后20s后服务器退出

- 创建定时任务: 20秒周期, 超时处理--》退出框架
- 添加时机: 玩家fini的时候若总玩家==1
- 摘除时机: 玩家init的时候若总玩家==0

## 随机姓名池

- 线性表存姓和名组成的线性表
- 取名字: 随机取姓, 随机取名
- 还名字: 尾部追加姓或名
- 读姓文件的同时读名文件, 边追加节点

## 守护进程

- fork关掉父进程
- 设置回话ID
- 重定向0 1 2
- 在 `/proc/XXXX(pid)/fd/` 目录中可以查到当前进程打开的文件描述符

## 进程监控

- 进入循环---fork
- 父进程, wait
- 子进程--》break

## 需求: 查看当前局游戏内有哪些玩家?



1. 创建文件（/tmp（存到内存的，重启会消失））存储当前游戏局的玩家们名字
2. 查询：显示文件内容
3. 设置：存姓名到文件或从文件中取姓名
  1. 存：追加的方式写文件
  2. 删：读出所有内容，将非自己的名字重写写入

## redis命令 (redis-cli XXXX)

- set key value: 存数据 (value)
- get key : 显示数据
- del key: 删除一对数据
- lpush, rpush存链表节点
- lrange遍历
- lrem删除n个节点

## redis程序结构

- cs结构，数据放在服务进程的内存中
- 命令行客户端连接本地或远程地址访问
- 多种API可以访问：hiredis
- 程序结构简单，内部的数据结构和算法优秀

## hiredisAPI使用

- C函数库，包含头文件 `<hiredis/hiredis.h>` ,编译时指定链接参数为 `-L/usr/local/lib -lhiredis`
- `redisConnect`跟数据库建立链接（`redisFree`释放掉）
- `redisCommand`发命令并通过返回值取出结果（`freeReplyObject`释放掉）
- 运行时若提示找不到共享库，则在`.bashrc`最末端添加一句 `export LD_LIBRARY_PATH=/usr/local/lib` ,重新打开终端运行

## 怎样写框架

---

## 面向对象的软件设计

1. 画用例图----》分析需求（不要考虑太多扩展，不要考虑实现方式）

## 回显功能的实现方式

- kernel类：基于epoll调度所有通道
- 通道抽象类：
  - 写出缓冲区函数
  - 将数据追加到缓冲区的函数
  - 虚函数：读，写，获取fd，数据处理
- 标准输入通道子类
  - 重写读和处理的函数
  - 处理数据的函数：将数据交给输出通道
- 标准输出通道子类
  - 重写写数据的函数
- kernel和通道类的调用
  - 创建通道对象（成员赋值）
  - 添加通道到kernel
  - run

## 添加FIFO文件通道支持

- 写FIFO类继承Ichannel
- 重写虚函数的过程中，重构抽象类Ichannel
- 添加构造函数的参数用来表示管道文件和方向

## 添加转大写功能

- 写新类（数据处理类）封装转换大写字母的功能
- 拆掉标准输入通道和标准输出通道的包含关系，在标准输入通道中包含数据处理类的对象
- 数据处理类中包含输出通道对象

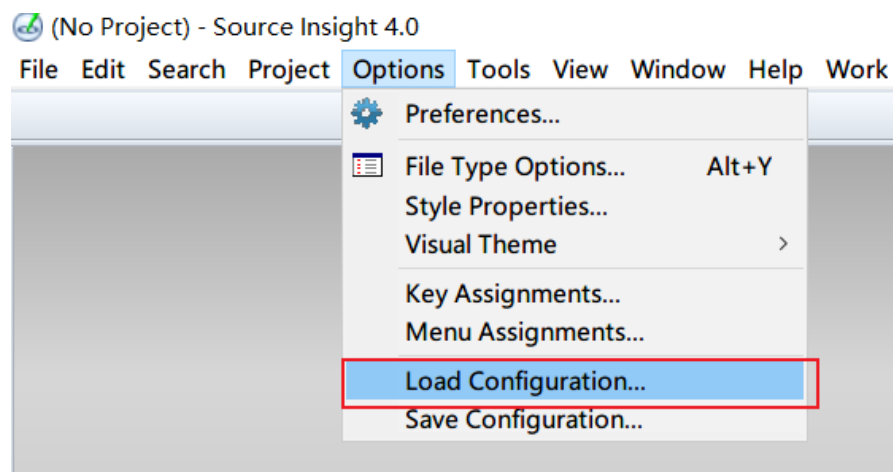
## 责任链模式

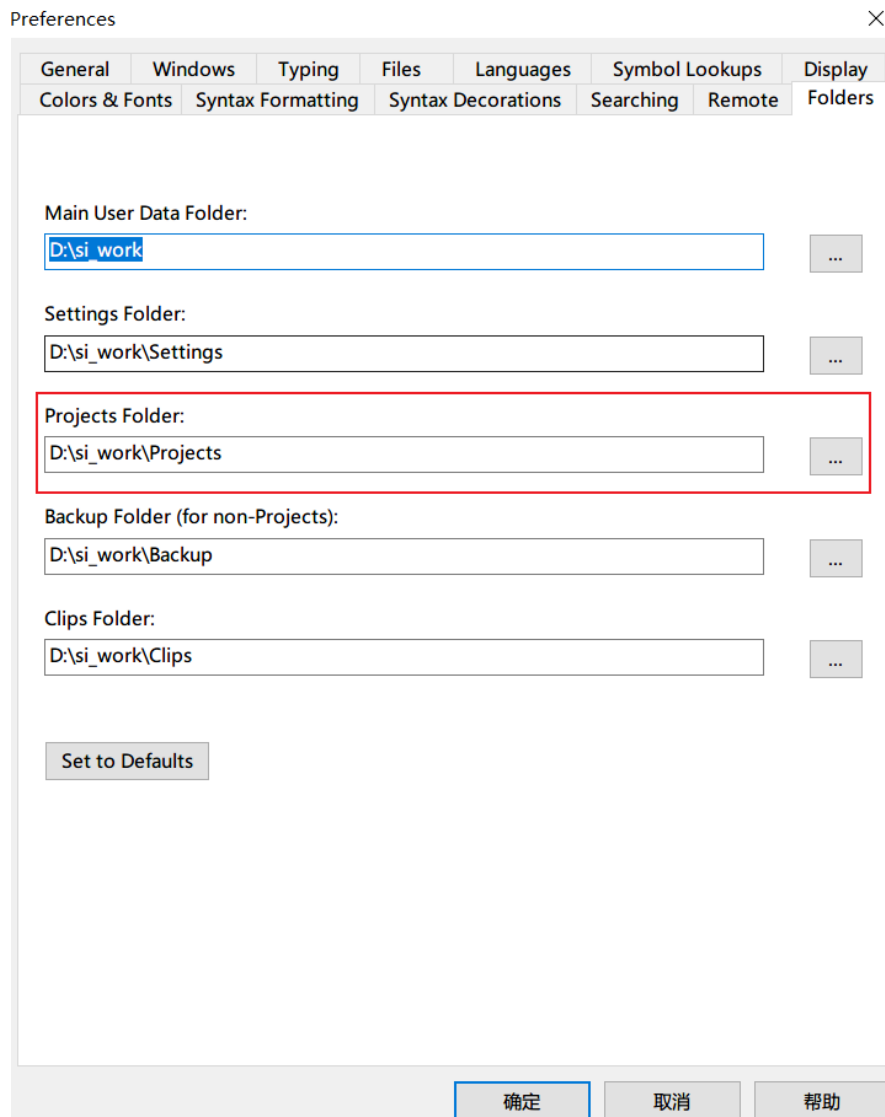
- 处理者类和消息类
- 处理者类需要子类重写内部处理函数和获取下一个处理者的函数

- 处理者类的外部处理函数：当前环节处理---》获取下一个环节---》下一个环节处理

## 重构当前代码

- 抽象通道类继承handler类，重写internal\_handle函数
- 定义消息类：IO方向和字节数据
- 功能处理类继承handler类
- 输入通道类getnext返回功能处理对象
- 功能处理类的internal\_handle 函数内直接调用zinx\_sendout输出数据
- 通道类，internal\_handle函数：
  - 消息方向IN，readfd
  - 消息方向OUT，缓存bytemsg对象中的content
- epollin事件：创建in方向消息--》交给channle的handle函数
- epollout事件：调用通道的flushout





## Sourceinsight快捷键

- ctrl+o 弹出文件选择栏
- alt+L 弹出符号栏
- ctrl+鼠标左键 跳转到函数定义
- alt + < 回退 alt+ > 下一个
- ctrl+1 显示调用关系
- shift+f8 高亮单词 ctrl+shift+f8取消所有高亮
- f7 查找符号

## 分发框架

- 库分发：编译成libXXXXX.so
- 编译参数：-fPIC -shared

- Makfiel中添加install目标，拷贝库文件和头文件到 /usr/lib 和/usr/include

```
sudo dpkg --remove cmake
```

```
sudo dpkg --remove libcurl4
```

## 容器技术

- 容器是操作系统和应用程序之间的一个虚拟层
- 应用程序可以在容器中运行（跟在操作系统中运行相同）。容器以应用程序的形式运行在操作系统中

## docker程序架构

- cs架构
- 容器，镜像都是由守护进程管理

## docker的三大核心概念

### 镜像

- 静态的一组环境的集合
- 运行：创建容器，在容器中运行XXXX
- 手动创建：
  - 下载原始镜像
  - 在基于该镜像运行bash，装所需软件
  - 将装好软件的容器提交为新的镜像
- 脚本创建：
  - 写Dockerfile：规定镜像创建的过程
  - 构建镜像

### 容器

- 运行时的一组环境，基于某个镜像创建
- 容器的修改不会影响镜像

- 运行容器：
  - 守护运行 -d
  - 端口映射: -p 外端口号:内端口号
  - 共享文件系统: -v 外绝对路径:内绝对路径
  - 容器开始于要运行的进程，结束于进程退出

- 删掉所有容器：

```
1 docker rm `docker ps -aq`
```

## 仓库 (dockerhub)

- 类似github，是一个存储镜像的公共仓库
- docker pull 作者/镜像名:标签名 拉去镜像
- docker push 分享镜像到仓库中（分享之前先改名--》docker tag）

## 脚本创建docker镜像

1. 指定基础镜像 (FROM XXXX)
2. 装环境 (RUN , WORKDIR, COPY)
3. 指定执行点 (ENTRYPOINT)

ENTRYPOINT ["XXX"]:

- XXX是且仅是镜像所运行的程序
- CMD 命令指定你的内容会作为XXX的参数
- run 镜像名 xxxx: xxxx会作为XXX的参数

CMD ["XXX"]

- 镜像缺省运行XXX程序
- run 镜像名 xxxx: 容器会执行xxxx程序

## 离线分发镜像

- 导出容器: docker export -o XXX.tar af85: 将容器中固化的内容导出
- 导入镜像: docker import XXX.tar my\_image:my\_tag: 导入的镜像只包含原容器内的文件系统，缺失了镜像执行点，暴露端口，原镜像的构建历史

## 查询程序依赖的动态库

ldd 程序名

## 静态编译游戏服务程序（部分依赖库静态链接到程序中的动态编译）

- 原因：不想在容器中装诸多依赖项
- 查当前程序的依赖项
- 添加所有 依赖项.a 到Makefile的依赖中
- 去掉Makefile中-l指定的链接库

## 游戏进程的启动脚本

- 首行#!/bin/bash
- 启动游戏服务的命令(./game)
- 死循环查询game进程是否存在
- 不存在则退出脚本

```
1  #!/bin/bash
2
3  #
4
5  [ a == a ]
6
7  p 10
8  ux | grep -w game | grep -v grep
9  } -ne $? ]
10
11  it 0
12
13
```

## 脚本创建游戏服务器镜像

- 有可执行的程序，有姓名文件，有启动脚本
- 指定工作路径
- 拷贝文件
- 暴露端口
- 指定执行点（执行脚本）

## 登陆服务器架构选择

- BS架构
- 效率要求不高，客户端向和服务端之间的数据交互比较单一：一条请求对应一条回复
- http协议承载数据

## http协议

- 超（基于字符编码多媒体）文本（字符串）传输（基于短连接tcp）协议
- 请求或回复头： 属性名：属性值
- http请求（浏览器---》服务器）
  - GET请求（只有请求头）
  - POST请求（头+身体）：头和身体直接有两个回车
- http回复（服务器---》浏览器）
  - 类型决定于请求类型
  - 包含头和身体
  - 头：状态码（200---》成功）
  - 身体：html文件（显示为网页）

## Nginx

- 概念：轻量化web服务器（高效的处理http报文）
- apt或者yum安装
- 配置文件目录：/etc/nginx/
- 日志配置文件：/var/log/nginx/
- 测试命令：nginx -t
- 重新加载配置文件：nginx -s reload

## nginx配置静态页面

- 创作一个html文件
- 将html文件放到server配置段指定的root目录下
- location配置段：index html文件名（指定首页）
- 浏览器输入192.168.64.148/XXXX.html----->服务器会向浏览器返回XXXX.html的内容

## nginx配置反向代理

- 规划url和对应的网页服务器
- proxy\_pass 被代理的服务；

## Nginx配置FastCgi

- 目的：跟客户端数据交互



## CGI程序处理流程

- 程序员自己写程序处理各种业务
- CGI程序的输入来自web服务器（标准输入和环境变量）
- web服务器会将http报文按照CGI标准转换后传递给CGI程序
- CGI程序返回http数据给web服务器（标准输出）
- 请求头：web服务器设定环境变量们传递给cgi程序，CGI程序通过getenv函数获取
- 请求的身体：web服务器直接将身体的字符串传递给cgi程序的标准输入

## FastCGI程序处理

- CGI程序缺点：资源消耗大，某些初始化操作冗余
- FastCGI：一共启一个进程，通过socket接收和回复web服务器传来的请求
- 数据是基于FastCGI标准编码过的二进制数据

## FCGI库使用

- 运行./autogen.sh + ./configure + make + sudo make install 安装

- 

```
1  (FCGI_Accept() >= 0)
2
3  获取数据(getenv--》请求头)
4  read ()  读请求的身体
5  处理数据
6  输出数据
7  printf 拼回复头和回复身体
8
```

- Fcgi库将socket数据转化成环境变量和标准输入交给{}内处理
- 编译后的程序既是FastCGI程序，也是CGI程序（可以通过命令行测试数据处理是否正确）
- 通过spawn-fcgi工具将程序运行成守护进程
- 监听端口要和nginx配置的端口相同

## 用户注册页面和处理

- 页面：点击提交后会向服务器发一个POST请求（身体是一串 用户名&密码 字符串）
- reg\_cgi程序，读取用户名和密码（fread 字符串处理）---》调用检查用户和添加用户的函数
- reg\_cgi程序，输出成功或失败的网页提示
- 添加用户函数：system调用添加用户脚本（不关心返回值时）
- 查询用户是否存在函数：fork+exec形式调用脚本（关心脚本执行是否成功）