BoyiaApp框架和伪进程研究方案

作者：晏波

一, 伪进程概念和机制

所谓伪进程, 即是指非真实进程，其作用范围仍然在应用程序自己的进程中进行，

其表现形式与进程相似，即隔离每个伪进程之间的内存容器，防止操作越界

同时为每个伪进程创建主线程，在一个伪进程中，使用boyia虚拟机所new出来的对象永远也无法超出该伪进程所开辟的内存范围，伪进程之间的通信使用伪进程间通信规则， 即共享内存。

二，伪进程实现流程

与Android系统类似：

1. 创建View DOM模型，该模型只要用于渲染，以及匹配CSS样式

该DOM的创建的内存使用伪进程提供的内存池

1. 创建boyia虚拟机，初始化虚拟机内存池，该内存池的创建源于伪进程内存

之上，是一块连续的内存。

1. 启动伪应用程序，boyia库使用App.main()来启动伪应用程序初始化，原则上

不会像AndroidThread一样一直loop，事件接受从Android Java inputEvent中传入。

三，伪进程间的通信（共享内存）

1. 共享内存原型，该共享内存也是一段内存池，维护一个HashTable，
2. HashTable中根据伪进程ID来维护一个消息队列，即每个伪进程都

在共享内存中拥有自己的消息队列，用于与其他伪进程之间的通信。

1. 伪进程要与其他伪进程通信，则需通过调用共享内存的接口，传入

指定要通信的伪进程的ID，即可完成通信，每个消息队列都是单工

数据传输。

四，boyia引擎设计思想

1. 引擎设计采用递归回调方式进行编译和执行，编译过程为递归模式

任何运算式都为一个Instruction结构，整个执行体为一个链式Instruction，

执行采用回调模式，过程为按顺序执行，即顺序调用Instruction所携带的handler进行执行。

1. 引擎对象采用哈希线性表存储其属性和成员函数指针，哈希线性表自身进行

对hashcode的排序（插入排序），在查找属性时采用折半查找节省查找属性时间。

1. 引擎JIT设计，借鉴谷歌V8中携带的简易汇编器进行改造和重新设计，主要完成了

ARM架构部分指令的拼接和封装, 将boyia instructions（即字节码级别的命令）转换为可执行二进制码缓存在内存中，加快执行速度。

1. 为加快引擎查询变量和对象属性的速度，有必要引入内联机制，内联机制原理：
2. 记录之前对象访问属性的索引位置
3. 不同对象的属性名一样且索引位置一样，即归并为同类
4. 是否为同一对象则根据其引用地址进行判断
5. 将动态判断的代码内联到主程序中，作为加速访问属性的依据

嵌入实例,伪代码

Instruction ins = {

{objStackIndex, ValType},

{inlineTable, ValType},

GET\_PROP\_INLINE, propHandlerInline};

解释一下：

1. 第一次运行时还是会走findprop流程，在整个属性列表中查找属性
2. 查找到位置后，会创建一个内联属性表，记录这个属性的位置和对象引用，

内联会记录访问该指令的做操作数对应的对象引用，以便下次查询，如果是相同对象则进行归并，如是不同对象，则重新查找

内联引用单元设计

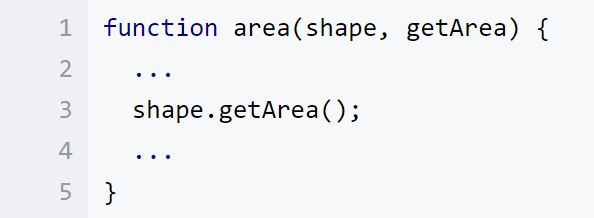
[ref | index]，通过index直接可以定位属性

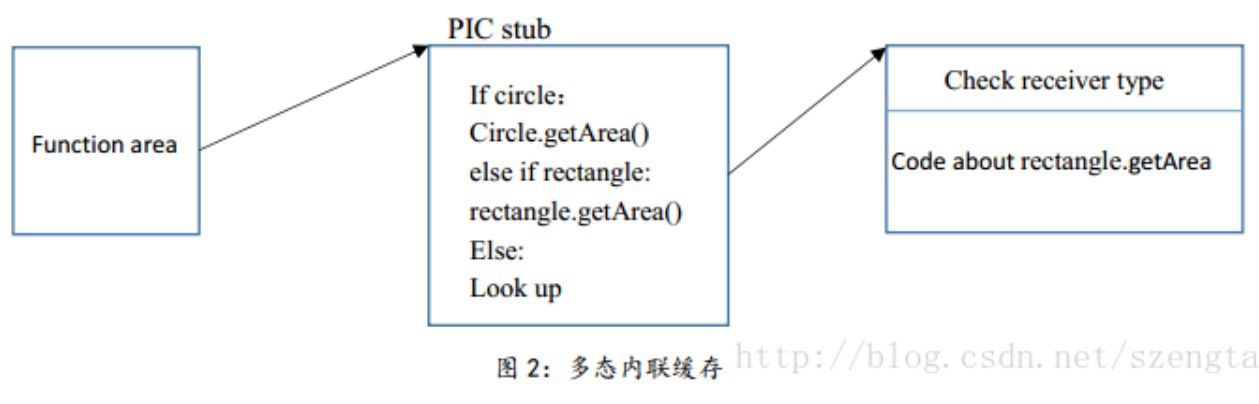
同一个指令可能对于多个单元

1. 创建新的Instruction, 指令类型为propinline，左操作数为对象，右操作数为内联属性表
2. 判断该对象引用是否存与内联属性表中，如果没有，则会执行下一步操作

，即查找属性，如果有，则利用传入的索引，定位对象中的索引，直接返回

Pc计数器指向下一行命令





1. BoyiaCore中类和对象模型，及其原因
2. 类设计

typedef struct {

LInt mPtr;

LInt mSuper;

} BoyiaClass;

2) 对象设计

Typedef struct {

LInt mRef; // 对象引用

LInt mType; // 类型，指向class结构

} BoyiaObject;

设计对象结构主要目的用于区别对象类型，使用instanceof()函数可以判断是否是从该对象实现或者派生

十一，参考Android APP系统启动流程

1，=>表示函数调用，2，->表示调用父类函数，3，{}表示完成过程

App\_process.main()=>{

AppRuntime.start()->AndroidRuntime.start()=>{

StartVm(), // 启动虚拟机

StartReg(),// 注册android函数

// 启动Zygote

ZygoteInit.main()=>{

RegisterZygoteSocket()，// 创建套接字，接受AMS的APP创建请求

// 启动系统服务

StartSystemServer()=>{

PackageManagerService,

ActivityManagerService,

InputManagerService,

WindowManagerService,

}

RunSelectLoopMode(),// 进入无限循环，等待Socket连接

}

}

}

十二，参考AMS启动应用程序流程

1，=>表示函数调用，2，->表示调用父类函数，3，{}表示完成过程,4[]参数列表

ActivityManagerService.startProcessLocked()=>{ // 启动实际应用

Process.start([android.app.ActivityThread,…])=>{

Process.startViaZygote() // 连接ZygoteInit的socket服务开启新的进程

}

}

IMS捕获event,交给wms，wms响应窗口事件，触发 launcher的点击事件，startactivity会调用ams的startactivity,ams使用localsocket通知zygote进程fork一个新的进程，反射调用activitythread main，将applicationthread对象attach到ams, ams通过pid对app进行初始化工作.

Evt start app fork reflect

IMS----->WMS----->Launcher----->AMS------>zygote----->ActivityThread

attach

main----->AMS

十三，参考Android事件派发流程

InputManagerService()=>{

NativeInputManager()=>{

EventHub(), // 初始化

InputManager()=>{

InputDispatcher(), // InputDispatcher初始化

InputReader([EventHub,…] // inputreader初始化

InputManager.initialize()=>{

InputDispatcherThread(),

InputReaderThread()

}

}

InputReaderThread.threadLoop()=>{

InputReader.loopOnce()=>{// 轮询调用eventhub来读取输入事件

EventHub.getEvents(), // 用于读取/dev/input/event0

// 封装event，然后交给InputDispatcher进行事件分发

InputReader.processEventsLocked(),

InputDispatcher.flush(),

}

}

}

}

Activity.dispatchTouchEvent()=>{

View.dispatchTouchEvent()=>{

View.listener.onTouch(), // 触摸监听

View.onTouchEvent() // 没有监听， 或者监听返回false，则执行

},

Activity.onTouchEvent() // 如果所有事件拦截都返回false，则该函数会被调用

}

Android其他相关知识锦集：

优化：

1. Activity对象尽量不要用static引用
2. Service相关知识

*1)BindService如果*服务没有启动，则会启动服务onCreate->onBind-> running -> onUnBind->onDestroy

*无论多少次bindservice，onBind都只会执行一次，但是会回调onserviceconnected, 即service被首次绑定时执行，如果activity没有主动unbindservice，会造成内存泄漏*

2) 当bindservice后，调用stopservice，然后关闭activity, 此时与service关联的context失效，也是会调用unbind和ondestroy

3) 多次*bindservice，然后unbindservice，只有最后一次调用unbind*

4) startservice->oncreate->onstartcommand->running->ondestroy

Bindservice->oncreate->onbind->running->onunbind->ondestroy

1. 减少view的层级，因为重新布局会从子元素递归到父元素
2. 减少主线程的工作量，任务较多则分担给子线程处理
3. 排序算法简述，顺序
4. 选择排序，[0]与[1…n]比较，[0]>[x]，两两交换，下次[1], [2…n]比较….
5. 插入排序，[0][1]比较排好序，[2]与[1][0]比较交换位置，依次
6. 冒泡，两两比较，最大到末尾
7. 快速排序，left=0,right=n, a=[0],right找比a小的位置[x1]，将[left]=[x1]，left找比a大的位置[x2], [right]=[x2]

如果left < right,继续，否则，[left]=a, 分而治之

1. 堆排序

建初始堆，大顶堆（顺序），小顶堆（倒序）

1. Androidstudio运行出现INSTALL\_FAILED\_INVALID\_APK，需要勾掉Instant Run
2. Dialog遮挡Activity是不会触发onPause之类的回调的, 原因在于dialog必须依附activity存在, dialog类似activity的view层，activity依然可见.
3. home，锁屏，关闭activity都会调用onstop
4. onActivityResult在onRestart之前调用
5. volatile告诉编译器该值不确定（不需要优化），需要重新计算，volatile并不能代替同步锁，因此在操作volatile修饰的值时，可能是一样的，但是可以作为状态标志使用
6. 动画：
7. 补间动画，
8. 属性动画，修改属性
9. Activity在singleInstance模式下会默认自带一个放缩的切换动，跟其他切换动画不协调，overridePendingTransition来替换这个动画
10. 启动优化：
11. Activity启动界面利用Fragment来做splash
12. Activity利用Window background来做splash，即android:theme=”@style/splash”
13. 透明或者设置为Dialog样式的Activity遮挡下面Activity不会调用onstop, 只会调用onpause
14. Context没有startActivityForResult

几个Context，Activity，Application

1）Toast可以利用ApplicationContext来显示

2）UI控件只能用Activity来作为context

3）getApplication和getApplicationContext区别，getApplication只存在于Activity和Service中，这时getApplication和getApplicationContext是一样的

4）Dialog只能使用Activity来作为context

1. android:multiprocess，表示Activity或者provider的实例能否被加载到与启动它的那个组件所在的进程中
2. Service和receiver启动activity时intent必须设置FLAG\_ACTIVITY\_NEW\_TASK
3. 在Activity中设置FLAG\_ACTIVITY\_NEW\_TASK默认不会启动新的栈，但是如果在Service中启动，则会创建新栈，原因是设置FLAG\_ACTIVITY\_NEW\_TASK后会判断是否和启动的Activity有相同亲和性的任务栈，有则把这个任务栈整体拉到前台，如果没有则新建栈。

但是，如果是两个应用中A应用的Activity启动B应用的Activity，intent设置FLAG\_ACTIVITY\_NEW\_TASK，则会新创建栈

整理：

1. IntentService,只是实现HandlerThread的一个Service，执行完后退出
2. Activity A->B->C, B设置FLAG\_ACTIVITY\_FORWARD\_RESULT，然后finish，可以在C中setResult
3. taskAffinity生效，launchMode需要设置成singleTask或singleInstance，如果不设置launchMode，Activity依然在原来的栈，但如果设置intent.addFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK)才生效
4. Handler和Looper解释，Looper的作用主要右两个，一是创建消息队列，二是轮询消息队列，Handler的作用主要是向消息队列添加消息，和实现处理消息的接口。
5. View绘制流程一般分为三步：

onMeasure,计算宽和高

onLayout, 计算坐标位置

onDraw，开始绘制

结合Android生命周期就是，onCreate-> View onFlinshInflate-> onStart -> onResume-> View onMeasure->View onLayout -> View onDraw->

onPause->onStop->onDestroy

小记1

View invalidate, requestLayout区别，invalidate只会调用onDraw，而requestLayout

则会调用onMeasure，onLayout, onDraw.

requestLayout会根据情况，如当前View位置，或者大小改变了，需要父容器重新布局，则会调用mParent.requestLayout()

小记2

View 绘制调用过程

Measure->onMeasure

Layout->onLayout

Draw->onDraw

1. android activity启动模式

standard，标准启动模式，没启动一次生成一个activity

singleTop，处于栈顶，则不会再次生成，如果不在栈顶，就重新生成一个

singleTask，不会再次创建，但是如果不是在栈顶，其上的activity就会被强制移除

singleInstance，拥有自己独立的栈，且栈中只有自己

1. android进程间通信，RPC，即远程接口调用

aidl，统一的接口，服务端binder实现这个接口，并实现接口提供的函数功能，

客户端通过调用接口来完成远程调用

1. Serializable, Parcelable区别，前者主要用在本地文件，数据库，网络流等

后者在内存中传递数据更高效为适应android进程间通信

1. SoftReference，具有强引用的特性，但是当内存不足的情况下，GC会将其回收，

比较适用于图片缓存。

1. ArrayMap数据结构，包含一个int[] mHashes,和Object[] mArray;

mArray存储键值对mHashes存储哈希，哈希查找采用二分法

根据hash的索引使用index<<1寻找key，使用index<<1+1寻找value

即2\*index，2\*index+1

1. SparseArray数据结构，包含int[] mKeys和Object[] mValues

Key值为整型，mValues存储value，查找Key使用二分查找，根据key的索引寻找value

10, Android优化

优化无非分为两种：

1. 时间优化

不同情况使用不同数据结构和算法，以及异步

如APP轮询查找固定数据时，最好先进行排序然后使用二分查找。

卡顿优化，如将上报等不常用的任务，单独开线程进行处理

1. 空间优化

内存优化分两种，内存泄露，内存碎片

减少循环引用，

对于频繁要创建的对象，使用缓存进行处理，避免频繁创建对象造成过度GC

11, GC分代，jvm gc算法实现

前言，jvm内存区域划分

eden, s0, s1, old

算法步骤

1)，在eden中new对象，

     分配内存之前检查eden区域，如果此时eden已满，则minor gc

     未清除的内存会从eden到s0（前提是s0为空），eden重置为空

2)， 继续在eden上new对象，如果eden再次检测到满了，

     那么，minor gc eden和s0，未清除对象移动至s1,

     s0和eden重置为空，同时交换s1和s0的指针，即s1变成s0

3)，继续在eden上new对象，如满重复2步骤，次数到很多次后，eden和s0移动未被清除的对象到old.

4)，当eden无法创建对象时(即便minor gc后依然无法创建)，则会执行fullgc，不同gc处理不一样，也有可能当old内存小于一定值时，此时eden依然能创建对象时执行fullgc

12, C++纪要。

1. new 和malloc的区别，主要在于new为操作符，可以自动执行构造函数，而malloc则不行，同理delete和free一样， delete可以自动执行析构函数，而free不行
2. 使用new在栈上创建对象(只是点placement new)，char mem[1024]; Object\* obj = new (mem) Object;
3. cmake 出现Fata signal 5(SIGTRAP)错误时，有一个原因就是有返回值的函数没有return
4. C++拷贝构造函数不能使用值传递，因为这样会造成递归调用，形成死循环
5. c++模板是两次编译，定义模板是进行一次编译，使用模板是进行一次编译，如果将模板定义放入cpp中，使用时只include模板的.h文件，则使用该模板的cpp将无法进行2次编译，因而无法编译出真正的实例。
6. C++标准明确表示，当一个模板不被用到的时侯它就不该被实例化出来；如果要将模板定义写在cpp中，则该cpp必须包含模板的使用，如：

template <class T>

void FunA(T t)

{

}

int main()

{

FunA(10); // call FunA(int) 编译器在这里决定给FunA一个FunA(int)的具体实现体。

}

13，LruCache维护一个缓存队列链表，其中队列的排列方式是按照访问顺序实现的，

即一直没有访问的对象将会放在队尾，即将被淘汰。

其实现是由linkedhashmap实现

14，java创建数组与C++的不同之处，C++数组是一块连续的内存，由数组名引用起始位置，java数组创建会先创建一个数组对象，数组对象中包含数组长度大小的对象引用，然后再去创建真实的数组堆内存，每个数组单元的引用地址指向该堆内存的某个地址.

15，JNI基本用法：

NewGlobalRef 全局引用，不能被垃圾回收，一直有效，只有当主动调用DeleteGlobalRef时被回收。

NewLocalRef 局部引用，能被垃圾回收，局部引用表最大不能超过512，必须调用DeleteLocalRef释放引用

只在native方法的上下文中有效，并且只在native方法的一次调用中有效，在该native方法未执行完之前，对象

不会被gc回收，当native方法执行完后，局部引用失效， gc可以回收该对象， 局部引用必须DeleteLocalRef来

删除本地引用，否则会造成引用表溢出。

NewGlobalWeakRef 弱全局引用，并被DeleteGlobalWeakRef摧毁，在native方法执行期间该对象不会阻止被垃圾回收

有JNI函数创建的大多数都是局部引用，如NewObject，NewStringUTF,NewByteArray等

16，LINUX fork进程

Fork函数执行一次，但是返回两次。

在fork函数执行完毕后，如果创建新进程成功，则出现两个进程，一个是子进程，一个是父进程。在子进程中，fork函数返回0，在父进程中，fork返回新创建子进程的进程ID。我们可以通过fork返回的值来判断当前进程是子进程还是父进程。

17，surfaceflinger系统原理,对GUI系统的深入研究

1）Surfaceflinger主要使用opengl进行绘制，opengl依赖本地nativewindow，

在Android中被命名为ANativeWindow

客户端SurfaceTextureClient->ANativeWindow

实际客户端实现ANativeWindow的为Surface。参看framework/native/include/gui/Surface

Surfaceflinger端FrameBufferNativeWindow

//////FrameBufferNativeWindow实现处于framework/native/libs/ui下

1. 客户端和Surfaceflinger是一种生产者和消费者关系，期间的数据传递

桥梁为BufferQueue, 客户端生产绘制数据交给BufferQueue，

Surfaceflinger作为消费方，从BufferQueue取出绘制数据

1. SurfaceFlinger结构介绍

GraphicPlane，用来描述显示屏，拥有一个DisplayHardware实例对象，

SurfaceFlingger的readToRun函数将对其进行创建和初始化。

DisplayHardware，主要工作是用于egl环境创建和初始化，与

FrameBufferNativeWindow相关

Layer，SurfaceFlinger中维护一组Layer，每个Layer对应一个app，

但是APP可以拥有多个layer，layer中包含了一个GLConsumer

作为消费者消费BufferQueue中的数据

FrameBufferNativeWindow的作用

[1], 初始化NativeBuffer，NativeBuffer是继承自

ANativeWindowBuffer，默认情况下NativeBuffer是两个

[2]，为NativeBuffer分配内存，使用gralloc

[3]，实现ANativeWindow对应接口，如下所示：

ANativeWindow::setSwapInterval = setSwapInterval;

ANativeWindow::dequeueBuffer = dequeueBuffer;

ANativeWindow::queueBuffer = queueBuffer;

ANativeWindow::query = query;

ANativeWindow::perform = perform;

在客户端Surface同样也实现了ANativeWindow,但其作为生产者进行操作

Surface介绍，服务端每一个layer对应客户端一个surface：

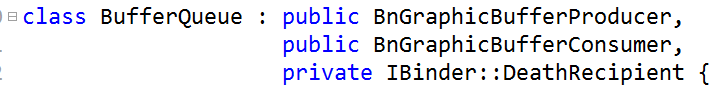
[1],包含一个GraphicBuffer，而GraphicBuffer却包含了ANativeWindowBuffer

[2], 包含一个数据生产者sp<IGraphicBufferProducer> mGraphicBufferProducer;

[3], mGraphicBufferProducer主要作用是和BufferQueue进行通信，为

BufferQueue在客户端的代理

[4]， BufferQueue实现了生产者和消费者接口，如下



整个UI系统的原理是，

Opengl接口在客户端作为生产者,生产图像数据，并交给ANativeWindow在客户端

的实现即Surface, Surface接连surfaceflinger中的BufferQueue，并将图像数据传输给它，服务端的Layer作为消费者，利用GLConsumer对BufferQueue进行操作

取出生产者生产的数据，利用服务端Openg进行归并整理生成纹理，输出到

ANativeWindow在服务端的实现，即FrameBufferNativeWindow来交互图像数据.

18，WMS系统研究

Activity端设计：

[1], ViewRootImpl 管理IWindowSession，

IWindowSession与WMS的Session一一对应

1. ViewRootImpl利用IWindowSession来创建W（窗口），如setView
2. ViewRootImpl利用IWindowSession来添加View，如addView
3. IWindowSession是在WindowManagerGlobal.getWindowSession方法中由

WMS的aidl接口IWindowManager创建

4）ViewRootImpl是DecorView的parent

[2],ViewRootImpl管理W, W实现了IWindow.Sub接口

与WMS的WindowState一一对应，在WMS中会将

W与WindowState进行关联，当修改窗口后会通知W.

1）ViewRootImpl同样由WindowManagerGlobal创建

[3]，IWindowManager为WMS在客户端的代理

[4], Window一个Activity拥有一个Window，其真实实现

为PhoneWindow，内含DecorView和AppToken

[5], WindowManager，WindowManagerImpl继承WindowManager，

1）WindowManagerImpl所有操作都是由WindowManagerGlobal来代为处理 的

2）Activity的getSystemService函数，并非获取系统服务的代理，如WINDOW\_SERVICE

只是简单的返回了WindowManager对象，而这个对象只是应用层管理对象

[6]，Activity组件在WMS和AMS之间的连接是通过AppWindowToken

来描述的。

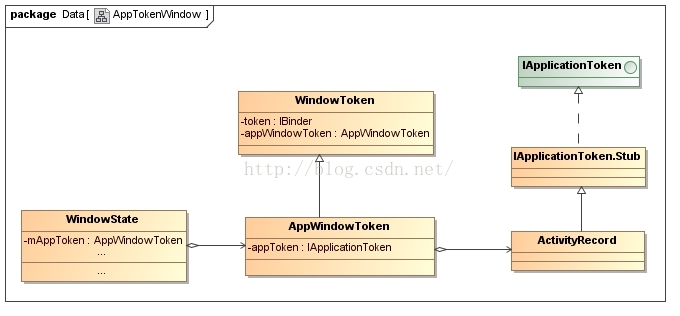
AppWindowToken继承自WindowToken

具体过程

<1> AMS为应用程序Activity创建ActivityRecord,该类在AMS内部，同时初始化一个IApplicationToken对象。

<2>AMS以IApplicationToken为参数向WMS请求创建一个

AppWindowToken



[6],每个Activity拥有一个Window

19，aidl RPC远程调用

StartService, 会走onCreate->onStartCommand

bindService,会走onCreate->onBind

[1], 多次调用startService，onCreate不会调用，onStartCommand每次都调用

[2]，多次调用bindService, onCreate和onBind都不会调用

如果客户端要取得与后台进程的联系，必须bindService，同时传入

ServiceConnection对象，

如果只调用StartService，关闭时只需要stopService

如果调用startService后再调用bindService，则关闭时需要调用unBindService，在调用stopService

1. java nio
2. java 线程
3. Thread.yield()，表示线程让出执行权限，并和其他线程一起再次竞争执行权限
4. Wait，wait表示等待唤醒，同时释放锁

十四，相关排序算法

1. 选择排序，即是选择一个元素作为基础点，然后在集合中查找对比，找出

最小值下标后swap，然后继续，算法如下：

void selectSort(int \*a,int n){

for(int i=0;i<n;i++){

int min=i;

for(int j=i+1;j<n;j++){

if(a[j]<a[min]) min=j;

}

swap(a[i],a[min]);

}

}

1. 插入排序，即先排序，然后后面元素在这个排好序的集合中查找合适自己的位置

如先对a0, a1进行排序，然后a2在[a0, a1]这个排好序的集合中根据比较来寻找自己的位置

一次类推a3,a4...

void insertSort(int \*a,int n){

for(int i=1;i<n;i++){

for(int j=i;j-1>=0&&a[j]<a[j-1];j--){

swap(a[j],a[j-1]);

}

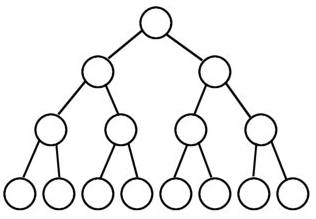
}

}

1. 希尔排序

4，堆排序

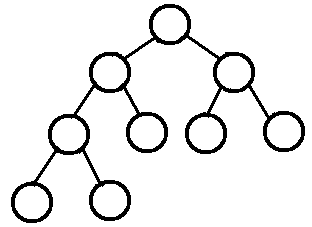
完美二叉树



即为满二叉树

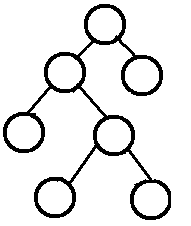
完全二叉树

完全二叉树从根结点到倒数第二层满足完美二叉树，最后一层可以不完全填充，其叶子结点都靠左对齐。

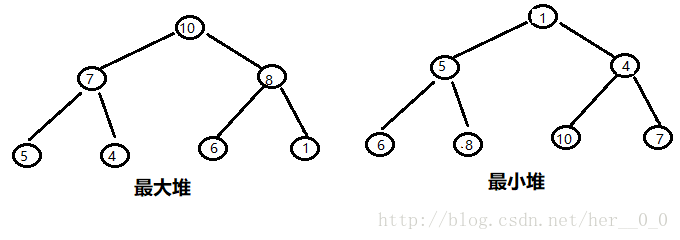


**完满二叉树**

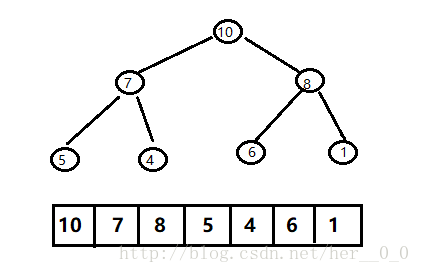
**所有非叶子结点的度都是2**。（**只要你有孩子，你就必然是有两个孩子。**）



**堆的定义：**   
堆实际上是一棵完全二叉树。   
堆满足两个性质:   
1、堆的每一个父节点都大于（或小于）其子节点；   
2、堆的每个左子树和右子树也是一个堆。   
**堆的分类：**   
堆分为两类：   
1、最大堆（大顶堆）：堆的每个父节点都大于其孩子节点；   
2、最小堆（小顶堆）：堆的每个父节点都小于其孩子节点；



**堆的存储：**   
一般都用数组来表示堆，i结点的父结点下标就为(i – 1) / 2。它的左右子结点下标分别为2 \* i + 1和2 \* i + 2。如下图所示：



算法如下：

void AdjustDown(int arr[], int i, int n)

{

int j = i \* 2 + 1;//左节点

while (j<n)

{

if (j+1<n && arr[j] > arr[j + 1])//子节点中找较小的

{

j++;

}

if (arr[i] < arr[j])

{

break;

}

swap(arr[i],arr[j]);

i = j;

j = i \* 2 + 1;

}

}

void MakeHeap(int arr[], int n)//建堆

{

int i = 0;

for (i = n / 2 - 1; i >= 0; i--)// n/2-1这个节点为最后一个元素的父元素

{

AdjustDown(arr, i, n);

}

}

void HeapSort(int arr[],int len)

{

int i = 0;

MakeHeap(arr, len);

for (i = len - 1; i >= 0; i--)

{

swap(arr[i], arr[0]);

AdjustDown(arr, 0, i);

}

}

十五，红黑树研究

1. 时间复杂度O(lgn)
2. 特性：
3. 根节点是黑的，
4. 叶子节点要么是NULL，要么是黑的，
5. 如果一个节点是红色，那么它的两个子节点都是黑的
6. 对于任意节点，其到叶子节点每条路径都包含相同数目的黑节点
7. 红黑树每次插入的节点都是红色，如果判断父节点也是红色就要根据

情况进行调整，如果插入的是左子树，就右旋，插入的是右子树，就左旋

十六，网络模型

1. ISO七层网络模型，其中TCP/IP为四层模型

应用层，如HTTP，FTP, RTSP

表示层, 如加密之类

会话层，如控制连接，心跳包协议

传输层，TCP，UDP之类

网络层，IP协议，路由器

数据链路层，如交换机之类

物理层，如集线器，网卡，网线，水晶头

TCP/IP协议总结：

1. 以太网最大传输单元MTU, 1518个字节， 负载1500个字节，18个字节头（以太网帧头14和帧尾4）
2. IP包头20，最大数据负载1480
3. TCP包头20，最大数据负载1460（MSS）
4. TCP三次握手，A->B, A询问B是否能够通信，B->A，B询问A是否可以通信，A->B，建立连接

TCP协议头包含，源端口Source port和目的端口Dest Port

1. UDP报文头部8，UDP最大负载数据大小为（IP包负载1480）-8=1472

鉴于Internet上的标准MTU值为576字节,所以我建议在进行Internet的UDP编程时.  
最好将UDP的数据长度控件在548字节(576-8-20)以内

6）另外还应该有个PPP的包头包尾的开销（8Bytes),那就为1492了

UDP 包的大小就应该是 1492 - IP头(20) - UDP头(8) = 1464(BYTES)  
TCP 包的大小就应该是 1492 - IP头(20) - TCP头(20) = 1452(BYTES)

1. 四次挥手，A(CLOSE\_WAIT)->(FIN)B, B(CLOSE\_WAIT)->(ACK)A, B(LAST\_ACK)->(FIN)A, A(CLOSED)->(ACK)B

通过四次挥手，可以把该说的话都说完，并且A和B都知道自己没话说了，对方也没花说了，然后就挂掉电话（**断开链接**）了 。

1. P2P下载设计思想：

NAT穿透技术，名词STUN简单穿透传输协议，TURN中继协议

1. 使用UDP技术进行穿透，STUN

一，客户端A向公网服务器发送节目请求

二，服务端收到请求，下发种子列表

三．种子列表中包含，已经下载过该节目，或者含有部分节目块的客户端列表，以及每个客户端列中描述的相关块号和块当前下载大小

四，A向客户端B请求，同时发送请求给服务器，要求服务器通知B，让B

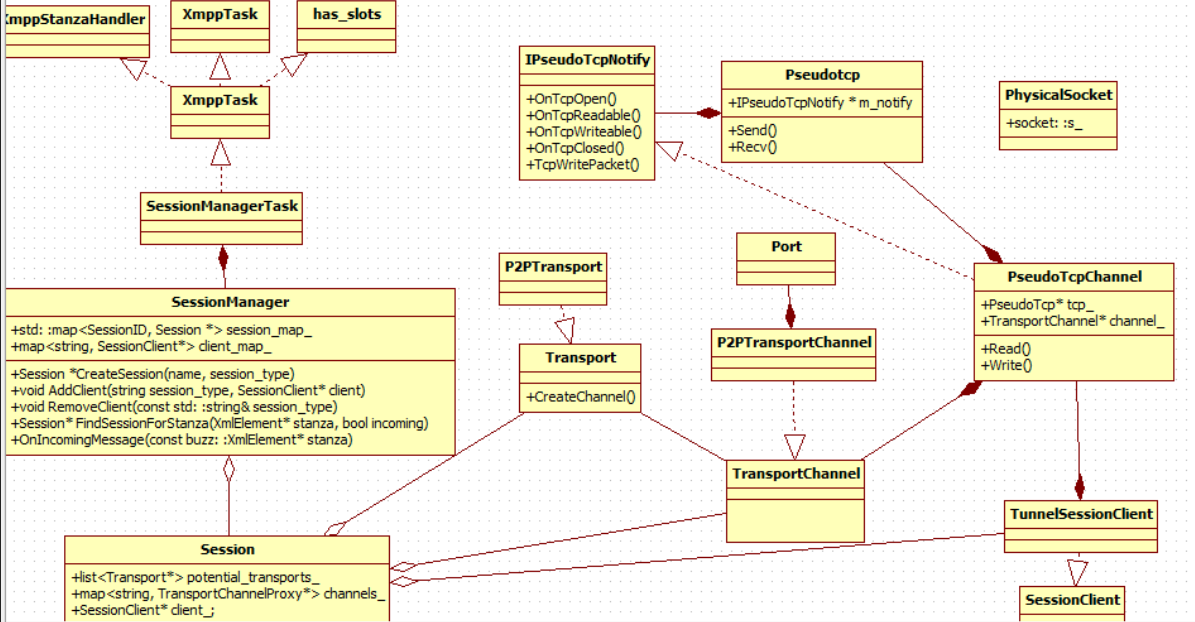
给客户端A发送确认请求，于是A和B就连接上了

2) P2P核心模块设计

1，libjingle采用了PesudoTcp来进行网络传输，我们抽取这个

模块进行开发和设计，PesudoTcp即是在UDP协议上的TCP

协议封装，由于UDP包长有限制，一般建议包长在548



**P2PTransport**:

是p2p数据体系上的抽象度较高的创建者和管理者。它创建和管理和销毁**P2PTransportChannel**，并且监视它的性能，但是实际上**P2PTransport**不处理数据；数据的真正进入点是**P2PTransportChannel**。

**Session**对象

掌管着**P2PTransport**对象，所以可以要求**P2PTransport**创建数据通道。尽管**Session**对象有掌管多个**Transport**和**Transport**子类实例的潜力，但是当前版本的代码只定义和使用了一个**P2PTransport**子类的实例。

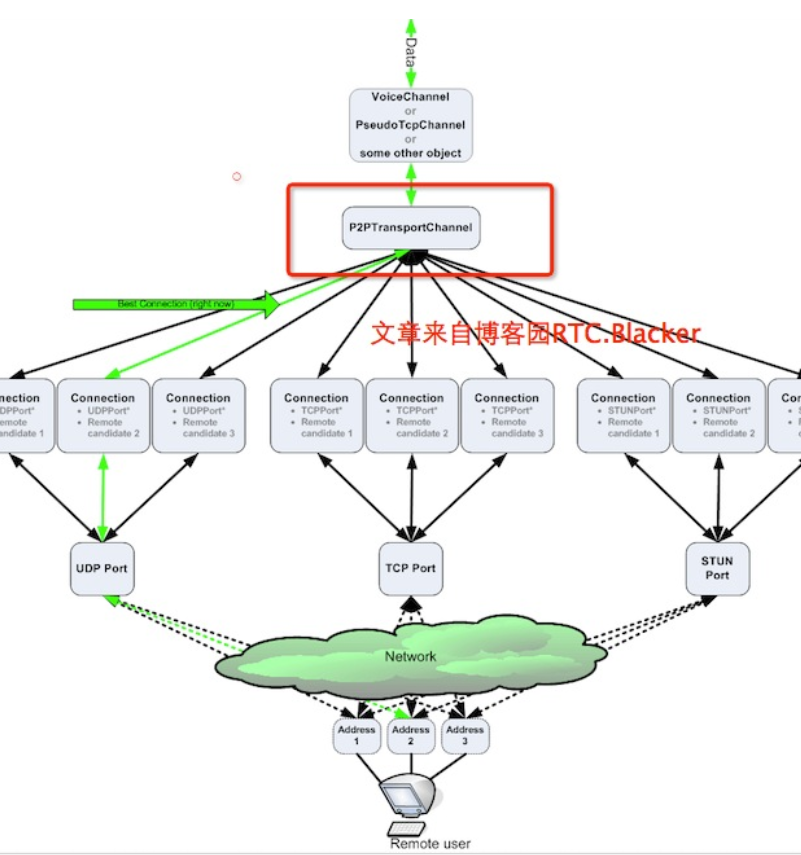
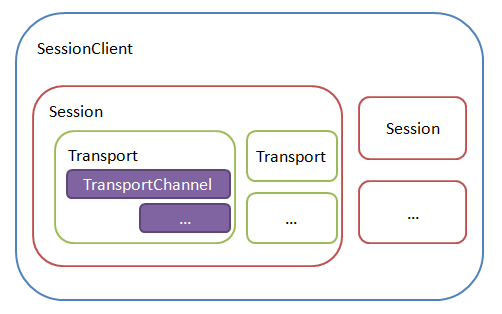
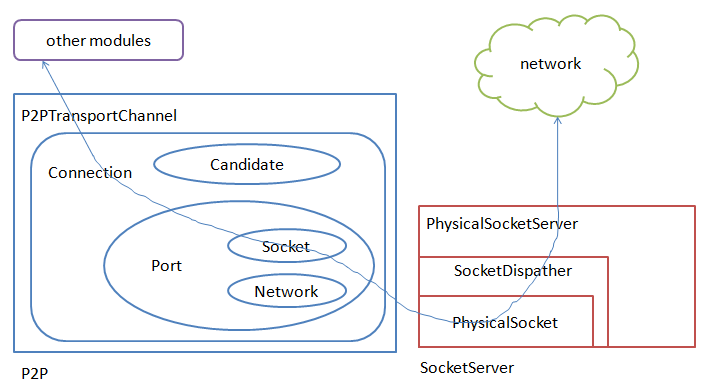
**P2PTtransportChannel**

创建和管理多个**Connection**对象。**P2PtransportChannel**根据可写入性和优先级（比如：UDP的优级先比中转链接的高）来选择最佳的**Connection**对象使用。

当链接异常中断或性能低下时，上述情况就会随之发生，**P2PTransportChannel**立刻会切换到下一个**Connection**对象上，并且这种切换对上层是透明的。

SessionManager

按content\_type来区分不同的sessionclient,两端的content\_type必须一样



十七，Opengl基础知识记录

1. VAO具体作用，以及与只使用VBO的区别

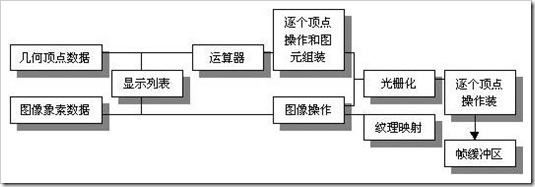
VAO主要是用来存储绘制信息的，用于加快绘制速度，其性能上比只使用VBO要好

VAO需要结合glVertexAttribPointer函数来进行操作

如使用VBO，则每次都需要指定glVertexAttribPointer的绘制方式

如加入VAO，则只需要一次绑定中加入glVertexAttribPointer指定绘制方式，下次重绘无需再次指定。

2，Opengl渲染流水线



1. 顶点计算
2. 图元装配
3. 投影
4. 光照
5. 光栅化，纹理映射，各种测试，输入帧缓存

如下为cocos2dx中对VAO的定义



VAO绑定，同时进行绘制：



C++，难点总结

1. 拷贝构造函数调用时机
2. Object a = b;
3. Object a(b)
4. 返回值是对象时
5. 函数参数为对象时
6. 拷贝构造函数不能使用值传递

如：Object(Object o) {}，这样会出错，使用后会进入死循环

Linux下安装Android SDK

1,下载sdk

wget http://dl.google.com/android/android-sdk\_r24.3.4-linux.tgz

1. 解压
2. tar zxvf android-sdk\_r24.3.4-linux.tgz
3. 进入tools，其中android文件为sdk命令

./android list sdk –all

查看sdk版本

1. 开始更新sdk

android update sdk -u --all --filter 1,2,3,5,11,12,22,23,24,25,26,27,28,29,88,89

Flutter应用环境配置

1. 运行Flutter目录下的flutter\_console.bat
2. 配置环境变量flutter\bin
3. 执行flutter doctor
4. 执行flutter doctor –android-licenses
5. 拉取第三方库

1）在pubspec.yaml中添加第三方库，执行flutter packages get

Maven安装

1. 下载apache-maven-3.6.0-bin.zip
2. 解压apache-maven-3.6.0-bin.zip
3. 配置环境变量，设置E:\Downloads\apache-maven-3.6.0-bin\apache-maven-3.6.0\bin
4. 进入项目目录，执行mvn help:system

Nginx负载均衡

