2020.06.12 温故

jni ndk编程

env：JNI函数表指针

但还是手动释放一下比较安全，因为在JVM中维护着一个引用表，用于存储局部和全局引用变量，经测试在Android NDK环境下，这个表的最大存储空间是512个引用，如果超过这个数就会造成引用表溢出，JVM崩溃

温习安卓知识, 学习python 串口相关的知识.

jni (06.12学习完成)

jni方法调用的时候类似python 方法的self参数一样,在c那一层将java调用层的实例当作参数放进来了.

JNIEnv是一个双指针链表,主要就是定义了一些JVM和c层桥接的方法, 例如findClass, Callmethod等等, 通过env调用的方法产生的都是局部引用, 8.0之前局部引用有数量限制, 8.0之后就没有512个局部引用的限制了.

jni里面的异常捕获在native和Java不一样, native发生异常不会直接退出, 而是继续往下走, jni的设计中, 只会保存一份最近的异常信息, 所以捕获异常显示或者return 之后, 如果后续还有异常会覆盖掉 上一个异常, 因此有异常需要及时处理, 并且处理的话也需要再处理完成之后clear掉 旧有的异常, 因为android只有一个JavaVM, env是从 JavaVM 中获取到的, 如果都没有虚拟机, 那谁给做find get set release工作呀.

jni主要是用于ndk编程, 如果有些高效计算, 或者音视频开发的话使用的比较多.本身是Java的规范, android有修改.

jni这方面做的不是太多, 现在主流的jni还要学习 cmake 这个编译链, 为了方便起见可以用ndk自带的, 如果要写通用的so 就需要原生的 cmake, 然后以不同的方式去编译输出. 这个不太会.

binder/parcel/aidl

核心是基于共享内存, 安卓的实现增加了安全校验, client和server的交互实际上都是和kernel的交互, BC是client/server 像kernel通过ioctl发送数据, BR是kernel对client/server的响应, client和server是彼此不知道彼此的存在的, server返回回来的数据就是内存的某一块数据, client 在收到数据之后就是直接拿着数据指针返回了, 在jni层翻译成bytes然后copy返回.

binder使用的特定的序列化parcel, 类似java的序列化, 但是不基于文件,而是基于内存的, 因此比较快速.效率比较高.

binder是基于binder驱动的, aidl是dsl语言, 根据一定的规则去翻译. 生成对应的BNBinder BPbinder.

android几乎所有的系统服务都是基于binder的, 安卓四大组件也都是基于binder的, messenger底层也是基于binder. 无binder没有安卓.

binder基于模板的代码, 会自动根据 宏 去生成一些通用代码. 我那时候用binder主要就是用的传递数据, binder的parcel数据是有特定格式和协议的, 我原来是因为那个exception 位置没有去判断, 导致数据翻译有问题.

binder里面的translate / ontranslate函数就是分发自定义的方法, 需要定义那个方法开始为0 , 然后在这个方法上递增就可以了, 声明 one_way flag的binder通信意思是你不用给我回复, 我就通知一下.

handler/looper/message

handler在post也各runnable或者sendmessage之后在他的实现里最终都会调用sendMessageDelayed, 这里最终调用的是queue.enqueueMessage 这里两个参数一个msg一个time, 这里最终调用的是MessageQueue里面, 这里会去根据这个是不是第一个如果是那就根据是否阻塞来决定是否应该唤醒, 如果不是第一个先把它查到对应的位置, 这个会判断是否阻塞, 以及当前节点的target(handler不为null)以及这个是异步的msg, 这时也会设定需要唤醒, 同时在插入的过程中会判断前节点是否也是异步的, 如果前面的节点也是异步的那么就会把needWake置为false, 最终判断为需要唤醒的话会调用

```
// 判断需要wake会调用这个 nativeWake方法, 如果不需要wake那么他就会在next里面阻塞一段时间, 然后就会超时或者取到event了提前唤醒, 然后下面就去处理这个mMessages, messageQueue实际上是链表实现的
if (needWake) {
                nativeWake(mPtr);
            }
// mPtr是创建native程序时返回的句柄
nativeMessageQueue->incStrong(env);
    return reinterpret_cast<jlong>(nativeMessageQueue);
// native层的对象转换为jlong了
```

这里的nativeWake实际上调用到了messageQueue的native层,

```
static void android_os_MessageQueue_nativeWake(JNIEnv* env, jclass clazz, jlong ptr) {
    NativeMessageQueue* nativeMessageQueue = reinterpret_cast<NativeMessageQueue*>(ptr);
    nativeMessageQueue->wake();
}

// 然后到这里
void NativeMessageQueue::wake() {
    mLooper->wake();
}

// Looper的wake方法
uint64_t inc = 1;
    ssize_t nWrite = TEMP_FAILURE_RETRY(write(mWakeEventFd, &inc, sizeof(uint64_t)));
// eventfd 是 Linux 的一个系统调用，创建一个文件描述符用于事件通知，自 Linux 2.6.22 以后开始支持
// 这里就是向mWakeEventFd写一个1

// 这里的mEpollFd监控的这个mWakeEventFd的变化
int result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeEventFd, & eventItem);
```

```
// Looper的构造方法里面调用rebuild初始化epollFd并添加对mWakeEventFd的监听
Looper::Looper(bool allowNonCallbacks) :
        mAllowNonCallbacks(allowNonCallbacks), mSendingMessage(false),
        mPolling(false), mEpollFd(-1), mEpollRebuildRequired(false),
        mNextRequestSeq(0), mResponseIndex(0), mNextMessageUptime(LLONG_MAX) {
    mWakeEventFd = eventfd(0, EFD_NONBLOCK | EFD_CLOEXEC);
    LOG_ALWAYS_FATAL_IF(mWakeEventFd < 0, "Could not make wake event fd: %s",
                        strerror(errno));
```

```
    AutoMutex _l(mLock);
    rebuildEpollLocked();
}

void Looper::rebuildEpollLocked() {
    // Close old epoll instance if we have one.
    if (mEpollFd >= 0) {
        close(mEpollFd);
    }

    // Allocate the new epoll instance and register the wake pipe.
    mEpollFd = epoll_create(EPOLL_SIZE_HINT); // 创建对应的fd节点
    LOG_ALWAYS_FATAL_IF(mEpollFd < 0, "Could not create epoll instance: %s",
strerror(errno));

    struct epoll_event eventItem;
    memset(& eventItem, 0, sizeof(epoll_event)); // zero out unused members of data
field union
    eventItem.events = EPOLLIN;
    eventItem.data.fd = mWakeEventFd;
    int result = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, mWakeEventFd, & eventItem);
    LOG_ALWAYS_FATAL_IF(result != 0, "Could not add wake event fd to epoll instance:
%s",
                        strerror(errno));

    for (size_t i = 0; i < mRequests.size(); i++) {// 这里第一次为空不走
        const Request& request = mRequests.valueAt(i);
        struct epoll_event eventItem;
        request.initEventItem(&eventItem);

        int epollResult = epoll_ctl(mEpollFd, EPOLL_CTL_ADD, request.fd, & eventItem);
        if (epollResult < 0) {
            ALOGE("Error adding epoll events for fd %d while rebuilding epoll set: %s",
                  request.fd, strerror(errno));
        }
    }
```

写了之后, epoll监听的对端就会有响应, 这时候就触发了去拉取的响应, 拉取是pollOnce最终也是调用的looper里面的方法.

更加深入的了解了一下这个机制.发现真的挺复杂的. 其中的nativePollOnce主要的功能就是去判断那个epoll的节点是否有事件产生, 产生之后他还会调用java层面的dispatchMessage去判断fd是否变了,如果变了且events有内容,那么就去增加这个fd到native的looper中, 如果events没有那么就从looper里面remove掉这个fd.

```
// 这里就是pollOnce的Looper调用, 他确定了requests里面没有需要的就调用pollInner, 在pollInner里面
再次去判断是否fd有变化等等
int Looper::pollOnce(int timeoutMillis, int* outFd, int* outEvents, void** outData) {
```

```
    int result = 0;
    for (;;) {
        while (mResponseIndex < mResponses.size()) {
            const Response& response = mResponses.itemAt(mResponseIndex++);
            int ident = response.request.ident;
            if (ident >= 0) {
                int fd = response.request.fd;
                int events = response.events;
                void* data = response.request.data;
                if (outFd != NULL) *outFd = fd;
                if (outEvents != NULL) *outEvents = events;
                if (outData != NULL) *outData = data;
                return ident;
            }
        }

        if (result != 0) {
#if DEBUG_POLL_AND_WAKE
            ALOGD("%p ~ pollOnce - returning result %d", this, result);
#endif
            if (outFd != NULL) *outFd = 0;
            if (outEvents != NULL) *outEvents = 0;
            if (outData != NULL) *outData = NULL;
            return result;
        }

        result = pollInner(timeoutMillis);
    }
}
// 在looper.c++的pollInner里面会调用这个,ident的token就是POLL_CALLBACK的时候,就调用一下
int callbackResult = response.request.callback->handleEvent(fd, events, data);
            if (callbackResult == 0) {
                removeFd(fd, response.request.seq);
            }
// 这里是调用java层面之后判断这个fd有变化
void NativeMessageQueue::setFileDescriptorEvents(int fd, int events) {
    if (events) {// 根据events类型增加不同的节点类型, 可能是input/output都监听
        int looperEvents = 0;
        if (events & CALLBACK_EVENT_INPUT) {
            looperEvents |= Looper::EVENT_INPUT;
        }
        if (events & CALLBACK_EVENT_OUTPUT) {
            looperEvents |= Looper::EVENT_OUTPUT;
        }
        mLooper->addFd(fd, Looper::POLL_CALLBACK, looperEvents, this,
                reinterpret_cast<void*>(events));
    } else {
        mLooper->removeFd(fd);
    }
}
```

当nativePollOnce走完了, 那么就不阻塞了, 这时候在java层就去读本地保存的mMessages变量, 然后进行操作, 这个mMessages变量是在enqueueMessage的时候确定的, 这个就是相当于头的message. 这里在messageQueue的next()里面拿到了需要的msg那么就设置mBlocked为false, 然后msg标记为markInUse然后返回, 这时候Java 层面的looper拿到这个message就去取它身上的target然后执行了, 这个target就是设置上去的callback. 没有就是默认的.

线程（thread）

线程的几种状态 sleep block running stop

死锁

死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上的资源。

查看分析死锁的时候，找出那些状态为BLOCKED的线程和他们等待的资源

ThreadLocal

用于保存线程特有变量，实现原理是在ThreadLocal里有一个ThreadLocalMap<ThreadLocal, Object>的成员，存储的值都在这个里面，她的key就是ThreadLocal，调用ThreadLocal的set get 等方法都是取出来这个Object，remove方法是将某一个ThreadLocal的键对应的值全部清空，ThreadLocal的Entry继承自WeakRefrence，当线程没有结束但是ThreadLocal被回收会有内存泄漏，因为共享变量还存在，没有释放，这时候需要调用remove去移除共享变量，或者使用ThreadLocal的定义为private static去规避这个问题。

get方法在获取变量的时候，如果没有设置，那么就从setInitialValue返回，这里是返回一个null，如果需要有默认值，可以复写initialValue方法。

为什么每个线程不重复

因为无论在set()或者get()都是先根据getMap(t) 去找到对应线程的threadLocal 这样肯定tnnd的取出来的是他的啊，日了狗，如果没有这个map对象那么就去创建一个对应的ThreadLocalMap 传进去的key不用说是咱的ThreadLocal，值么是initialValue提供的，默认的是null，**因此刚刚有在主线程把threadLocal当作构造参数传递给子线程，导致在run方法里面设置不生效的问题，nnd是因为Thread.currentThread获取的就是主线程了。**

安卓里面常见用的地方

Looper里面使用ThreadLocal保存Looper变量，因为Looper实现类是静态的方法等，所以使用ThreadLocal去为不同的HandlerThread保存Looper变量。

原子类

AtomicBoolean

AtomicInteger

AtomicRefrence

LockSurport

java实现锁的基本单元，内部是调用的native方法。

UncaughtExceptionHandler

Java捕获非检查异常，静态方法 setDefaultUncaughtExceptionHandler() 为应用里的所有线程对象建立异常 handler 。

阻塞队列

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

Callable，Future和FutureTask

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象, 使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

阅读这篇文章了解更多关于Callable，Future的例子

FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读Java FutureTask例子，学习如何使用它。

ConcurrentHashMap

http://ifeve.com/java-concurrent-hashmap-2/

http://ifeve.com/concurrenthashmap-weakly-consistent/

协程

python协程就是存有方法区的堆栈数据等的放入队列里，当协程方式执行的方法发现自己是耗时的操作，例如i/o或者网络的话，那么就主动让出cpu，然后python会将这段调用栈（栈帧）放到队列的尾部，如果大家都是这样那么都在往尾部加，然后程序再去调度调用那个，当然里面会有多久之后唤醒他，那么程序自然也会在多久之后去唤醒的，这个可以做成类似handler的。

线程池

ThreadPoolExecutor, 线程池原理, 主要是几个不同的worker去共享一个任务队列. 阻塞的去共享。

java.util.concurrent.Executors提供了一个 java.util.concurrent.Executor接口的实现用于创建线程池。线程池例子展现了如何创建和使用线程池，或者阅读ScheduledThreadPoolExecutor例子，了解如何创建一个周期任务。

对象池(Meaasge)

安卓的Message里面有一个pool这个池子的长度是5, 提供池子的目的是减少内存分配与内存波动，因为Message不仅仅继承是parcel还有一些对象在里面。

设计模式（0708）

基本六大原则

接口隔离原则（最小化接口范围，细化接口，可以实现多接口，这样调整一个接口的影响最小），面向接口依赖倒置（面向抽象而不是实现），低迷特（最小知道原则），**单一职责（一个类只干一件事情），开闭原则（对修改关闭，对扩展开放），李四替换原则（李四的儿子能够变成他爹，因为他有他爹所有的遗传）**

单例

整个软件生命周期只有一份，这个一构造函数私有，有一个静态内部类（静态内部类构造出静态的对象),提供一个公有静态方法获取静态内部类生成的对象

监听者

就是各种listener，例如安卓里面的onclicklistener

观察者模式

就是拥有observer的那种模式

适配器

listview的各种adapter将各种元素翻译为另外的元素

Proxy模式

提供一个proxy实现，可以在构造方法里传入被proxy的对象,然后proxy实现相同的接口，就是对接口声明的方法做一些加工或者包装，代理模式

工厂模式

工厂模式，抽象工厂和简单工厂，抽象工厂不提供一样的实现，简单工厂就是提供一个create方法，每次new都可以。

责任链模式

（view的事件传递），如果有角色处理了这次点击事件返回true，就相当于他消费或者处理了这次事件，尽到了责任

拦截器模式

（filter对不满足一定条件的进行拦截，一般就是一个数组，包含对象,判断对象是否是自己要得不是则去掉）

生产者/消费者

一般是两个线程，生产者线程生产的东西就放到一个队列，然后消费者去消费这个队列就可以了，例如handler那个就是典型的生产者模型，你无论在那个线程post task或者send msg到队列，那个looper所在的handlerThead就是消费者，拿到msg就消费啊

状态机

在阵列项目中用到的一种架构方式，为了解决耦合问题，不同的状态对应不同的执行，写成状态机的形式有助于扩展，另外可以控制状态的转移，但是我们的状态以及状态转移都比较单一，没有那么复杂。

Builder模式

当参数特别多的时候使用这种模式，这个模式的好处还一个就是可以链式调用，可以有默认值，用起来方便。

不用builder也可以使用链式调用去做builder类似的效果，但是他俩还是不一样，这个暴露类

常用数据结构

hashMap：

基于散列表的Map内部是Entry数组

arrayMap：

安卓实现的Map，扩张的时候是加4还是多少，不是double的那样增长

ArrayList：

基于数组的list，基于数组需要连续分配内存，查找简单，删除需要copy

linkedList：

基于链表的list，寻找比较麻烦，插入删除简单

ConcurrentHashMap：

http://ifeve.com/java-concurrent-hashmap-2/

自带片段锁的Map，线程安全

Queue：

队列先进先出，一般task 或者 数据里面用的比较多

Stack：

先进后出，括号的解析等用的栈比较多

常用算法

快排

https://wiki.jikexueyuan.com/project/easy-learn-algorithm/fast-sort.html

快排核心是问题分解，使用两个移动下标，一个最左侧一个最右侧，如果从左边开始，那么就应该首先移动右边的找到比这个基准值小的，然后左边的那个游标找到第一个大于基准值的，他两个交换，然后继续移动直到碰头，碰头了就把碰头的值和基准值交换位置。

然后分解子问题，将第一个基准值分割好的两侧再次使用同样的算法分割重排，直到只剩下一个值那就不需要排了。

二分法查找

```
int BinarySearch(int array[], int n, int value)
{
    int left = 0;
    int right = n - 1;
    //如果这里是int right = n 的话，那么下面有两处地方需要修改，以保证一一对应：
    //1、下面循环的条件则是while(left < right)
    //2、循环内当 array[middle] > value 的时候，right = mid

    while (left <= right)  //循环条件，适时而变
    {
        int middle = left + ((right - left) >> 1);  //防止溢出，移位也更高效。同时，每次循环
        if (array[middle] > value)
            right = middle - 1;  //right赋值，适时而变
        else if (array[middle] < value)
            left = middle + 1;
        else
            return middle;
        //可能会有读者认为刚开始时就要判断相等，但毕竟数组中不相等的情况更多
        //如果每次循环都判断一下是否相等，将耗费时间
    }
    return -1;
}
```

hash

就是一个散列表算法，为了防止数据碰撞的，例如HashMap的的hash算法为：

```
return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
```

lruCache

是一个线程安全的数据结构，因为在get set 等方法里面有加sync锁，基于LinkedHashMap双向链表结构做的一个算法，最近使用在尾部，最多使用在尾部，在set的时候会判断同样的key是否有被添加过，如果有就会把之前的删除掉，然后回调entryRemoved去做内存回收。核心算法在trimToSize里面去判断是否超过长度，超过就从前往后去删除他。put方法调用的时候塞进去然后判断之前是否有值有的话就释放了他，get方法就是如果有就命中，如果没有就调用create方法，这个应该是用户去复写的不然就是返回null。

afterNodeAccess方法是linkedHashMap提供的，负责将最近访问的数据放到尾部。

AMS

负责四大组件的调度生命周期的管理等，我们在做vulcan的时候就是在ams里面插桩防止互拉，主要的做法就是在允许的时候返回true我们返回false。

PMS

做VR的时候在这里做的白名单功能，在判断这个包名是符合一些规则的或者包含的组件有某些关键字规则的情况下会去把这个包加入白名单，然后在渲染的时候不开VR效果。

另外在做vulcan云控的时候通过反射调用pms的方法完成应用的冻结，卸载，隐藏等。

Input service

android输入模块，通过驱动产生事件，然后在服务启动时候创建拥有两个socket的bittube，然后在驱动读到数据，例如坐标值等等就通过一端发送给另外一端，这里收到这些时间就回去解析。

input会产生event down / up 等事件，还有多指头花屏就是把这些点存到了一个数组里面。

研究这个就是为了做虚拟键盘，将时间touch的坐标映射到3D环境中，然后如果产生碰撞了那么就算touch到了某个键值。

双击唤醒的一种实现

就是两次tap的间隔时间小于某个阈值，例如100ms，那么就判断是双击，同样的长按事件的产生也是类似的，这里是通过使用一个handler发送一个delay task这个task到时间了就去check一个变量是否还是一个摁下的状态如果是那就触发长按效果，如果中间有up的事件产生就去handler里面remove掉这个task。也就不会产生长按事件了。

SurfaceFlinger (基于opengles)

surface flinger的研究路径是从最简单的开机动画开始，这里就是直接拿到的opengles去画画和渲染的。然后主要学习了surface的大致渲染流程，结合自己写的一个系统服务，这个服务就是拿到陀螺仪数据，然后通过掉一个jar包把这些数据通过binder传给surface这一层，在这里渲染的时候我会拿到生成的texture然后把这个扔给3D SDK，他们对这个texture进行渲染和使用。我的主要工作就是拿到这个合成的最终的画布，但是使用硬件渲染的情况下拿不到，那时候做到使用opengles方式渲染然后达到3d效果。

应用启动流程

https://juejin.im/post/5e8b64f6518825737a314b7c

1. 启动进程，点击launcher会调用对应的startActivity
2. 开启主线程app，主要是实例化ActivityThread创建 ApplicationThread， Looper， Handler对象, 并且开始消息循环
3. 创建并初始化Application和Activity
4. 布局绘制

开机启动流程

1. 启动linux内核启动init进程，这个是负责很多工作，类似开机动画，启动zygote进程，启动serviceManager
2. fork zygote，他的工作是创建一个server端socket和客户端进程通信，加载类和资源，启动system server，监听socket（**这里之所以用socket是因为binder都还没有启动**）当有应用启动时会通过socket发出请求，然后zygote fork自己创建子进程
3. systemserver进程，主要功能为启动binder线程池，初始化looper，创建system service manager 他启动各种服务，例如ams pms wms，启动桌面，开启looper循环，开启消息循环
4. zygote运行之后在启动app都是zygote的子进程，达到复用资源的效果。

IPC Binder(进程间通信)

对于一个Binder接口，在客户端和服务端各有一个实现：Proxy和Native，它们之间的通信主要是通过transact和onTransact触发。一般从命名上可以区分：xxxNative是在本进程内的Binder代理类，xxxProxy是在对方进程的Binder代理类。

ServiceManager管理所有的Android系统服务，有人把ServiceManager比喻成Binder机制中的DNS服务器，client端应用如果要使用系统服务，调用getSystemService接口，ServiceManager就会通过字符串形式的Binder名称找到并返回对应的服务的Binder对象。

它是一个系统服务进程，在native层启动，它在system/core/rootdir/init.rc脚本中描述并由init进程启动。

ServiceManager启动后，会通过循环等待来处理Client进程的通信请求。

```
// binder大小限制为1M - 8k
#define BINDER_VM_SIZE ((1 * 1024 * 1024) - sysconf(_SC_PAGE_SIZE) * 2)
```

aidl

binder通过aidl来进行binder的一些方法的调用和声明，就是一个糖，对binder来说，这个就相当于一个模板，系统会帮助解析成对应的binder b端和s端，就是那些模板代码没有意义还必须有，所以为了使用方便给加了这个机制。aidl里面会有 in out oneway等。

序列化

java是实现 Serializable 接口， 这个是有磁盘i/o所以效率比较低

安卓有一个 parcel， 这个是内存序列化，设置某一个值为1的时候也会有磁盘i / o

可以使用google 的 proto协议

proto

```
syntax = "proto3"; // 声明使用的版本

package com.example.studyproto;// 包名
option java_outer_classname = "MyInfo";// java bean的名字

message Info {
    string account = 1;// 第一个成员变量
    string password = 2;// 第二个成员变量

    enum ABC{// 枚举
        AB = 0;
        BC = 1;
        CD = 2;
    }

    message InnerInfo {// 内部类
        string num = 1;// 内部类的第一个成员变量
        ABC a = 2;// 第二个成员变量
    }

    repeated InnerInfo iinfo = 4;// 外部类的第四个成员变量
}

message AllInfo {
    repeated Info info = 1;// 成员变量
}
```

使用proto可以用来网络通信，也可以用来我们程序之间的通信，proto也是实现了序列化的协议，因此如果直接调用writeTo到文件里，读出来的时候还是可以通过parseFrom input数据流里面解析出来存储的文件数据。

proto对二进制有压缩，所以理论上使用proto传输数据可以更加节省宽带。

I/O多路复用 select / epoll

为什么主线程 looper 死循环没有anr,因为处于block状态, 而不是没有响应, 如果给对端写一个激活那就有响应了，阻塞状态下也会放弃cpu的使用权，只有等待的资源有了响应才会去征用cpu所以不会anr

这两个是i/o多路复用的实现，select有缺点就是她的效率略低，需要遍历通知，有fd限制，一般是1024个，epoll因为每一个事件是基于callback的效率高直接找到接收这个消息的进程，epoll基于链表长度也没有限制。

epoll就是三步，第一创建, 第二调用epollc_ctrl注册，第三调用epoll_wait等待事件的产生，这个是阻塞的，当下的程序后面只能等阻塞通过，但是不会消耗cpu资源。

网络 http 80 https 443 websocket

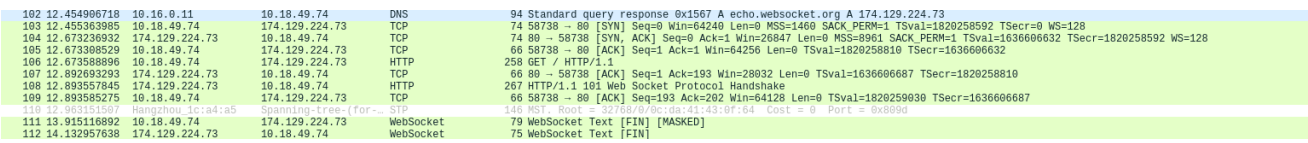
http默认使用的端口都是80端口，https则是443端口，另外还有stmp，ftp，telent， 等等都是用的小于1024的端口，也就是所谓的知名端口号，websocket是谷歌基于tcp的上又封装的一层协议，就是在header里面加上信息去确认这个是websocket，这个协议是全双工通信的协议，有一套类似的回调接口，这样就能够在server这个协议的好处是实时性比较好。

> http://www.ruanyifeng.com/blog/2017/05/websocket.html
>
> 其他特点包括：
>
> （1)建立在 TCP 协议之上，服务器端的实现比较容易。
>
> （2）与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽,能通过各种 HTTP 代理服务器。
>
> （3）数据格式比较轻量，性能开销小，通信高效。
>
> （4）可以发送文本，也可以发送二进制数据。
>
> （5）没有同源限制，客户端可以与任意服务器通信。
>
> （6）协议标识符是ws（如果加密,则为wss），服务器网址就是 URL。

websocket

建立连接及通信

websocket是一种和http同一层级的协议，使用了http1.1的初始化连接，在连接建立之后，会发送一个头去通过upgrade告知对方变为websocket协议，具体流程如图：



首先是三次握手建立TCP连接，这时候紧接着就会调用http1.1的get方法，在headers里面会设置upgrade为websocket然后也会声明host等等。等到server端同意了之后会用http1.1返回这次请求的结果在headers里面声明请求的信息以及响应的操作，这时候ws算是建立起了连接。接下来的websocket协议就是通信了。

断开连接

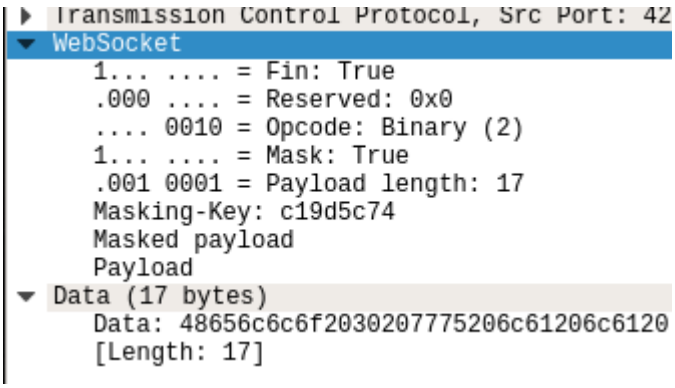
断开连接过程如图，

38	7.311779351	10.18.49.74	174.129.224.73	WebSocket	74 WebSocket Connection Close [FIN] [MASKED]
39	7.517741823	10.18.49.74	203.205.179.168	TCP	74 57988 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=260222229 TSecr=0 WS=128
40	7.532922280	174.129.224.73	10.18.49.74	WebSocket	70 WebSocket Connection Close [FIN]
41	7.532969710	10.18.49.74	174.129.224.73	TCP	66 59466 → 80 [ACK] Seq=240 Ack=233 Win=64128 Len=0 TSval=1821368633 TSecr=1636884078
42	7.532986857	174.129.224.73	10.18.49.74	TCP	66 80 → 59466 [FIN, ACK] Seq=233 Ack=240 Win=28032 Len=0 TSval=1636884078 TSecr=1821368412
43	7.533279969	10.18.49.74	174.129.224.73	TCP	66 59466 → 80 [FIN, ACK] Seq=240 Ack=234 Win=64128 Len=0 TSval=1821368634 TSecr=1636884078
44	7.607046301	10.18.49.74	239.255.255.250	SSDP	213 M-SEARCH * HTTP/1.1
45	7.757208184	174.129.224.73	10.18.49.74	TCP	66 80 → 59466 [ACK] Seq=234 Ack=241 Win=28032 Len=0 TSval=1636884133 TSecr=1821368634

首先是通过websocket协议说我要断开连接，然后server响应：哦，好吧，那就断了。接下来就是TCP标准的断开连接四部曲，首先是client向server说撒有那拉（seq=240 ack=233），然后server回复(seq=233 ack=240)，然后client再说(seq=240 ack=234)，然后server回复(seq=234 ack=241)这时候才会真正的关闭socket。

为什么要有四次挥手，就是保证他们之间可以和平分手，不然还有流没有发送完毕收到一次就直接关闭太粗暴，c端先说我要断了啊，s端会看有没有数据了有的话抓紧给到c端，然后说断吧，c端说断了啊，s最后说好的我这边没有给你的了我也释放资源了，最终断掉。

头格式



分别为fin（如果是1表示是数据的最后一片） reserved（三个0值一般情况下，可以扩展含义） opcode（这个值决定怎么解析后续的数据载荷） mask（是否对数据做了掩码操作） payload length（数据载荷长度）mask-key（所有从客户端都会进行掩码操作，这些字节不包含在payload length里面），payload data（x+y字节，包含载荷数据 扩展数据 应用数据）

socket编程 udp/tcp

再写python版本的配置工具sdk时大量的使用了socket方面的知识。socket是网络里面应用层的东西，屏蔽了一些细节让我们可以开心的使用基于tcp/udp进行网络编程; tcp/udp是运输层，给数据加上对应的tcp或者udp头; ip/icmp（ping工具就是这个协议）/ igmp是网络层，给数据加上mac地址; 数据链路层则是路由器数据帧以太网数据等等

使用socket主要是因为往阵列里面写入配置，这个设计的协议是发现设备的时候通过udp发送广播，发送广播之后我拿着我发送广播的那个socket在阻塞等待响应消息的到来，有了我就解析并加入到list里面，tcp是我往里面写入证书配置的时候用的协议，这个就是用的selectors去阻塞等待响应，因为我是一个tcp的client，没有办法通过timeout去，然后通过selectors去监听这个socket身上的read事件，如果server有响应了那么我就从这个socket上面去把这些信息读出来，判断是否写入成功还是超时了。

写入配置这个有一个问题就是写入之后我内部会发送一个停止配置的的tcp包过去，这时候板子会在15s之后重启，因此如果成功了立刻去配置就会有问题，可能配置到某一步失连了导致我的socket失效，也可能连不上等等，但是我有了socket又导致没有分配新的socket给他。

还有一个基于串口的配置工具一对一的我么有写还，实际上和socket类似只是需要设置码率，然后通过write和read去控制，几乎类似了。

语言方面

python

使用python3写过几个sdk，包括配置工具sdk， 事件上报server（redis，requests，也包括文件上传），工厂测试工具sdk，与后台server交互的sdk，包括登录注册拉取数据等等

写一个python server，restful风格的server，使用的是flask栈，主要用到的有之前的使用redis的sdk，以及使用mongoDB去保存和读取数据，里面有封装的数据结构，也有通过python3以上版本内置的线程池，以及selectors模块去阻塞通知等等（之所以用这个是因为我的一些东西是通过redis publish过去的需要等待一段时间，我就用这种形式去阻塞等待响应，有响应了对socket一端写入，我这边响应）

使用python写了一个类似安卓handler机制的工具，主要构成部分为 **message**（里面包含一个handler的引用，一个task可以为空）， **task**（类似runnable）， **handler**（负责handle message如果msg的task字段不为空那么就执行这个task）， **handlerThread**（继承thread，run方法里面去new一个looper然后调用她的loop方法）， **looper**（构造方法里面new一个自己的message queue主要是放到线程里去loop，就是无限等待message queue里面有没有新的msg产生），**message queue**（**主角**，持续等待别人post task或者send msg进来，task也会描述为msg，这里主要用到的是selectors epoll或者select机制，也是一个读端一个写端有事件了就会去写端写一个唤醒阻塞等待的读端一侧，这里的核心算法有两处一个是**next**方法这里是在loop里面调用的方法，这里就是调用的时候首先设置阻塞时间为0s去读一下有没有事件如果有那就判断这个事件到时间了吗没有那么就把msg里面的时间减去现在的时间，在进入阻塞等待状态。另外一个重要的方法是**enqueueMessage**方法，这个方法就是将msg塞到对应的位置，并且去判断是否需要唤醒阻塞，在唤醒之前需要把设置mMessages这个节点）。这个工具是轻量级的，与安卓的不同有很多，安卓的毕竟使用场景比较多，很多都在native去做的唤醒等等，然后因为安卓的msg继承了序列化接口以及其结构较为复杂所以安卓用了对象池减少内存频繁分配，增加复用性，我这里的message没有这样做。looper 安卓是静态工厂创建的我是new， 在message 里面安卓还有一手messgener进程间通信在，这个底层是基于binder的，message queue也可以设置不同的fd去给到安卓做唤醒节点，安卓里面直接通过eventfd去创建了一个节点，有事件直接写一个1,读这个的就能知道了。而我用的是socket getSocketPair（基于TCP的一对socket，所以不能timeout，因为本身一直是阻塞等待的状态）去拿到两个socket。同样使用epoll达到了这个效果。

java

面向对象,Java属于高级面向对象的语言，语言特性详细分析在这篇里面。主要包括高级语言的特性

封装

数据和方法封装到一个类上

多态

（1）方法的重载和重写

（2）子类对象的多态性

使用前提:a.有类的继承 b.由子类对父类方法的重写

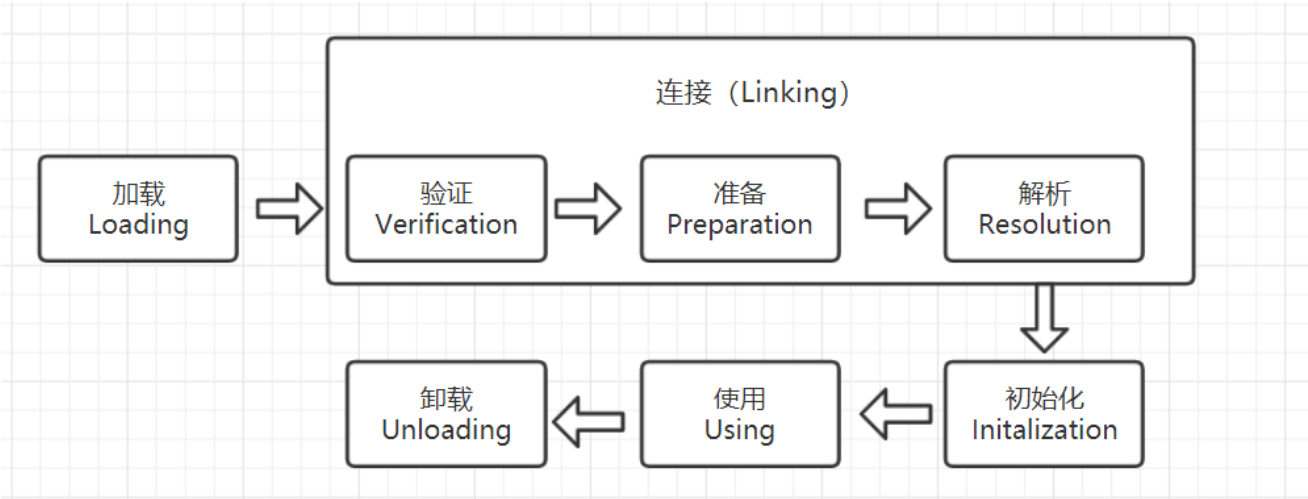
继承

子类可以继承父类，并且可以再次实现父类的public和protect方法，继承类还可以继承父类的数据。

抽象

拥有抽象能力，可以将现实事务提取用高级语言描述出来。

JVM加载运行字节码流程



加载负责找到对应的字节码流文件，验证保证是合法的符合规则的，准备和解析是把类的方法，常量，父类，使用到的别的类解析下，初始化调用cinit（有的没有，没有静态变量和静态代码块），init（构造函数），使用就是加载到内存中的了，卸载就是从JVM中移除。

kotlin

kotlin大略可以看作是加了糖的Java，所谓的解决null pointer其实就是通过各种糖强制提示你的，如果非得弄一个null 然后!!照样空指针。

然后kt比Java高端的就是协程啥的，但是Java最新的好像这部分也快有了，kt用过简单的，太深入的也不了解

dart

类似js的语言啊，用这个写flutter最别扭的就是不写xml去配置ui，全部是代码去控制ui啊，这就导致了各种缩进等等，但是看人家ali某个flutter控件的源码，人家就封装的比较好，写的条理也清晰。重要的还是思想。

c / c++

写过少量的native代码, 那个时候是将陀螺仪数据计算成四元素, 是java计算, c / c++ 使用的一种场景, 同时也包含了binder的使用. 其实我那个时候的价值就是获取数据 ,给到 opengl 使用, 他们拿着这些数据去画对应的姿态

shell/bat

使用shell bash编写一些辅助脚本。这些就是从网上找到基本用法，直接按照自己需要的来救可以了，主要就是$0 $1，if then fi

powershell

刚刚学会用powershell写一些可执行脚本，这个后缀是ps1，这个感觉更像shell可比bat那种好用多了啊哈哈。基本上就是谷歌，木的办法。本来是查找bat调用，后来在strackflow发现别人说ps1本来就是一个脚本语言，为啥还要用bat调用，bat就是傻啊。

```
# 如果要执行ps1脚本，需要以管理员运行并且设置 set-executionpolicy remotesigned
rm -Force -Recurse .\build\
rm -Force -Recurse .\dist\
```

```
rm -Force Main.spec
echo 'remove old success '
sleep 1
# 新增的add-data用法
pyinstaller --icon=producetool.ico --add-
data="fighter_widgets\static;.\fighter_widgets\static" --add-
data="server_sdk;.\server_sdk" -F .\producetool\Main.py

echo "first build success, will update spec, and build."
echo ''
echo ''
$content = Get-Content -Path '.\Main.spec'
echo $content
sleep 2
# 这里替换内容的是[]是特殊字符需要反斜线隔离
#$nc = $content -replace "datas=\[\]", "datas=[('fighter_widgets\\static',
'.\\fighter_widgets\\static'), ('server_sdk', '.\\server_sdk')]"
#echo 'after spec is :'
echo ''
#echo $nc
#$nc | Set-Content -Path '.\Main.spec'
#E:\Program\py37\Scripts\pyinstaller.exe -F Main.spec
$t = Get-Date
$tt = $t.ToString('yyyy-MM-dd HH-mm-ss')
$name = "ProduceTool-$tt.exe"
echo "build success $t"
# 对一个文件或者文件夹重命名
Rename-Item 'Z:\Test\python\fighter-desktop-ui\ProduceTool\dist\Main.exe' -NewName $name
# 进入一个文件夹
start 'Z:\Test\python\fighter-desktop-ui\ProduceTool\dist'
```

javascript

一点点类似dart，通常是因为单线程，里面有协程？

groovy

学习过groovy 语言，是因为学习gradle配置学习过一些，不甚精通，只知道一个闭包等等，但是因为知道这个是java系的语言，所以本质上是相通的，它是一门DSL语言，领域内使用的。

音频

aac格式

将pcm变为aac格式再上传为了节省宽带，aac是一种不保真的音频压缩格式，不同的采样率16k 44100k不同的编码格式

wav格式

只是比PCM多了44字节，头里的44字节保存音频的采样率，位深（short int 等等），时长，大小等等

视频存储(包含简单编解码)

将视频流使用原生的codec压缩成mp4格式存储, 主要涉及简单编码, 空间管理, 各种异常处理, 一些命名取巧, 按天分文件夹

视频聊天(使用的janus/web rtc)

janus 主要是协议的封装使用, rtc负责解析流并渲染到对应的view 上.

应用升级模块

与后端对接

headers的约定, rc4 sha1 加解密, url 生成, body加密, base64编码等, 这些都属于网络呢感觉..

阵列产品

pcm数据大小端, 按照协议实现udp 数据发送, 整体阵列项目架构, 里面分为executor, statemachine, xmos audio 录音模块, uploader, command , doctors, doa .

该产品主要功能就是通过xmos模组获取几路数据,然后根据这些数据使用fftw提取特征值, 然后做一个相似度的计算, 根据相似度去对比是否为异常声音, 后期增加为3阶相似度, 为了更加精确的匹配音源特征.

角度是为了确定发声的位置, 主要是利用4路或者6路数据去立体确定音源位置.

App运行流程为 启动后 收到命令或者执行内置命令开始录音, 对录到的声音做一次加工,就是拿着200ms的数据去计算特征值相似度这些, doa等, 拿着这些附件的描述放到对应的数据结构中, 然后拿着特征是和相似度去比较是否异常, 原来是0.93 作为标准的, doa 是取某段时间内出现最多的角度, 角度是瞬时的,所以取他的平均值, 实际上看那个代码, fftw这个也有平均值因为一次就是20ms, 200ms是取的十份特征的平均值.

redis, docker, fastdfs, mongoDB

redis分布式

redis 订阅发布，分布式使用

docker 容器化

docker 以及 docker-compose使用，达到固化开发环境，以及给客户一键使用

fastdfs文件上传等服务

fastdfs文件管理

mongoDB分布式存储

mongoDB 分布式存储，这个主要是类似json的形式去存储，读出来都能直接用，注意使用的时候要加上对应关键字段的索引。

conda

python 环境模拟，当然也有别的作用，但是我主要用conda去创建一个独立的python环境去做编译等工作

常用开发工具

android studio

开发安卓必备，谷歌基于idea开发的，好用

pycharm

idea家的python开发工具，开发python好用

vs code

开发工具，各种语言，强大的插件库

meld / beyond compare

文本比较，主要是比较两个文件的不同

yed

流程图

dia(流程图)

画流程图或者类图

wireshark 抓包

通过抓包工具分析是否正确的收发数据

vim

vim编辑器

postman

模拟网络请求，get和post最常用的把

大端和小端

大端又称 **高尾端** 网络传输数据默认是 高尾端

```
// 举例 存入的数据为 0x11223344 那么大端表示为

// 低地址 -> 高地址 其中0x01为起始地址
11   22   33   44
0x01 0x02 0x03 0x04
// 由此可见 大端比较符合第一直觉 就是尾部的数据放到了高地址的位置, 因此叫做 高(地址)尾(尾部数据)端
```

小端又称 **低尾端**

```
// 举例 存入的数据为 0x11223344 那么大端表示为

// 低地址 -> 高地址 其中0x01为起始地址
44   33   22   11
0x01 0x02 0x03 0x04
// 由此可见 小端反人类, 低地址却放着大数据, 因此叫做 低(地址)尾(尾部数据)端
```

进程和线程

进程和线程简单而基本靠谱的定义如下：

1. 进程：程序的一次执行（**进程资源天生独立，因此有了IPC通信**）
2. 线程：CPU的基本调度单位（**线程资源天生共享，因此有了多线程编程/同步/ThreadLocal等**）

同步（互斥量、条件变量、读写锁、文件和写记录锁、信号量）

mutex，一句话：保护共享资源。

semaphore的用途，一句话：**调度线程**。

锁优化

锁粗化（Lock Coarsening）：也就是减少不必要的紧连在一起的unlock，lock操作，将多个连续的锁扩展成一个范围更大的锁。

锁消除（Lock Elimination）：通过运行时JIT编译器的逃逸分析来消除一些没有在当前同步块以外被其他线程共享的数据的锁保护，通过逃逸分析也可以在线程本地Stack上进行对象空间的分配（同时还可以减少Heap上的垃圾收集开销）。

轻量级锁（Lightweight Locking）：这种锁实现的背后基于这样一种假设，即在真实的情况下我们程序中的大部分同步代码一般都处于无锁竞争状态（即单线程执行环境），在无锁竞争的情况下完全可以避免调用操作系统层面的重量级互斥锁，取而代之的是在monitorenter和monitorexit中只需要依靠一条CAS原子指令就可以完成锁的获取及释放。当存在锁竞争的情况下，执行CAS指令失败的线程将调用操作系统互斥锁进入到阻塞状态，当锁被释放的时候被唤醒（具体处理步骤下面详细讨论）。

偏向锁（Biased Locking）：是为了在无锁竞争的情况下避免在锁获取过程中执行不必要的CAS原子指令，因为CAS原子指令虽然相对于重量级锁来说开销比较小但还是存在非常可观的本地延迟（可参考这篇文章）。

适应性自旋（Adaptive Spinning）：当线程在获取轻量级锁的过程中执行CAS操作失败时，在进入与monitor相关联的操作系统重量级锁（mutex semaphore）前会进入忙等待（Spinning）然后再次尝试，当尝试一定的次数后如果仍然没有成功则调用与该monitor关联的semaphore（即互斥锁）进入到阻塞状态。

IPC

本地的进程间通信（IPC）有很多种方式，但可以总结为下面4类：

消息传递（管道、FIFO、消息队列）

管道就是pipe这种

共享内存（匿名的和具名的）

https://juejin.im/post/5e2986bce51d454d7a259d6d

和binder一样，都是系统有一个对应的驱动设备，通过对这个设备的操作来达到共享内存的效果。

/dev/ashmem

```
[java] android.os.SharedMemory#create
[jni] /frameworks/base/core/jni/android_os_SharedMemory.cpp#SharedMemory_create
[libc] /system/core/libcutils/ashmem-dev.c#ashmem_create_region
[driver] /drivers/staging/android/ashmem.c#ashmem_open
```

ashmem_mmap

`ashmem_mmap`通过调用内核中`shmem`相关函数在tempfs创建了一个大小等于创建`ashmem`时传入大小的临时文件（由于是内存文件，所以磁盘上不存在实际的文件）,然后将文件对应的内存映射到调用mmap的进程。

通过mmap申请一块共享内存，两个进程可以一起使用

远程过程调用（Solaris门和Sun RPC）

RPC远程过程调用，类似别的地方调用另一个地方的代码片段,执行结果也返回

Socket通信

这个应该天生就是跨进程的通信方式，input模块传递事件用的就是基于这个的bittube。

Binder

binder就是基于共享内存的，binder有大小好像安卓给的限制是1M大小，所以如果有大内存的数据进程间交换最好还是用共享内存，例如图片什么的，安卓在binder上做的东西很多，基本上安卓就是围着binder转的，所有的系统服务都是binder的，都是哪一个binder句柄,然后调用方法。

当然binder的实现还是很复杂的，有模板，宏定义，智能指针，aidl（这种就是一个语法糖，屏蔽了许多同样的代码，也减少了出错几率），oneway代表有去无回，就是我调用一个方法不用给我返回值，类似void？？

JVM（Java虚拟机）

强引用，软引用，弱引用

https://www.jianshu.com/p/e5364c05cc80

强引用：

就是我们普通的new 指向的引用，这里虚拟机GC的时候，如果可达，永远不会回收

软引用：

这个需要SoftReference包一下，这里在内存吃紧的时候会回收这个对象,如果内存充足那么就不会回收

弱引用

她的生命周期最短，只能存活到下一次GC之前，这里是通过WeakRefrence包的

上述三个引用的生命力一次减弱，回收的时候都是调用的finalize方法，弱引用别回收之后，其弱引用本身也被放到引用队列里，然后GC的时候删除掉这个引用。

引用队列：

ReferenceQueue清除失去了弱引用对象的弱引用本身, 软引用,虚引用也是如此.

字节码

coffe-baby

抽象类，接口等实现

接口与抽象类的区别：

1. 抽象类具有部分代码实现，提高代码的可重用性
2. 一个类只能继承一个类，但是可以实现多个接口

- 行为模型应该总是通过接口而不是抽象类定义，所以通常是优先选用接口，尽量少用抽象类。
- 选择抽象类的时候通常是如下情况：需要定义子类的行为，又要为子类提供通用的功能。

Java异常机制

https://www.cnblogs.com/yangming1996/p/8954700.html

Java中所有异常继承自Throwable，其中直接继承自throwble的error平常程序中少用到，另一类是exception在程序中比较常见，而exception的子类，又分为两大类，一个是i/o相关的受查异常，如果有这些异常那么编译器都会强制提醒要捕获，另一类是runtime异常，这个只有在运行的时候才会产生，故而不是受查异常。

自定义异常，只要继承对应的异常即可，抛出异常可以通过在代码里面throw也可以在方法上通过throws抛出异常。

其中 try / catch / finally finally中的逻辑无论如何都会走到，因为反编译字节码指令会发现，finally有两份指令复制一个在try 块 一个在catch 块，无论是否在try / catch中return，finally里面的语句都会执行，因为字节码把它放到了return前面，如果finally里面有return 那么就不会执行try / catch 里面的return了。

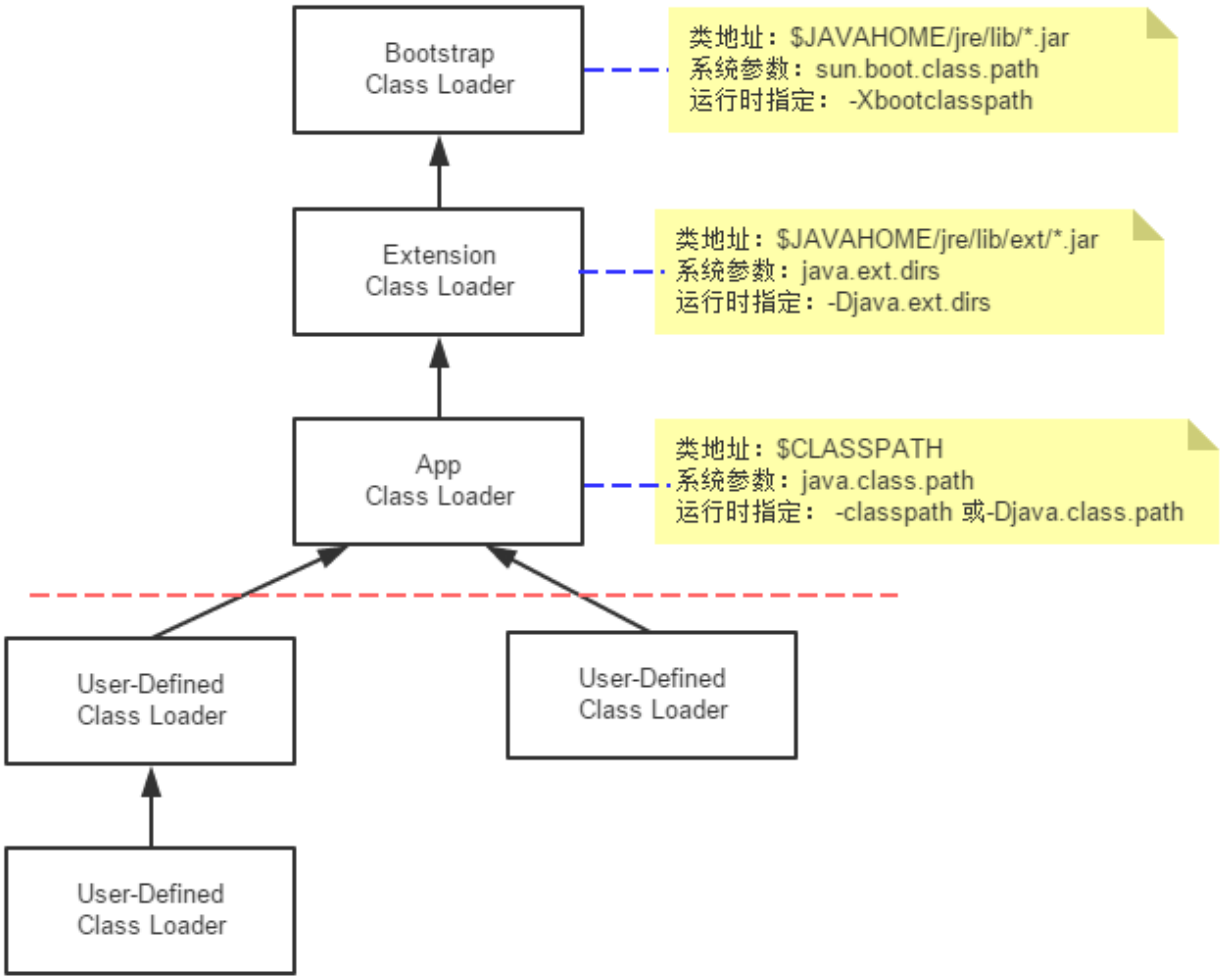
异常消耗资源

这个问题要从 JVM（Java 虚拟机）层面找答案了。首先 Java虚拟机在构造异常实例的时候需要生成该异常的栈轨迹，这个操作会逐一访问当前线程的栈帧，并且记录下各种调试信息，包括栈帧所指向方法的名字，方法所在的类名、文件名，以及在代码中的第几行触发该异常等信息，这就是使用异常捕获耗时的主要原因了。

Classloader

https://www.cnblogs.com/significantfrank/p/4875795.html

这个其实通过看 深入理解Java虚拟机 学习过现在忘了，通过上述网站完成速记。



双亲委派

classloader双亲委派模型，就是倒序加载，如果用户有自定义的classloader那么就用自己的，否则用App的，app的是定义在CLASSPATH的位置的文件，如果还没有就从EXT找，最后是Bootstarp找。如果这里都找不到那就抛出NotFound异常。

1. 委派的意义：主要是安全性考虑，在JVM里面只保存一份字节码
2. 委派是否必须：不是必须的
3. 实现双亲委派：默认的loadclass是双亲委派，如果要实现那就在复写class loader的时候重写findClass
4. 破坏双亲委派：复写loadclass

ContextClassloader

线程的一个属性，使用这个机制可以打破双亲委派

反射，注解

https://blog.csdn.net/briblue/article/details/74616922

反射

1. Java 中的反射是非常规编码方式。
2. Java 反射机制的操作入口是获取 Class 文件。 有 Class.forName()、 .class 和 Object.getClass() 3 种。

3. 获取 Class 对象后还不够，需要获取它的 Members，包含 Field、Method、Constructor。
4. Field 操作主要涉及到类别的获取，及数值的读取与赋值。
5. Method 算是反射机制最核心的内容，通常的反射都是为了调用某个 Method 的 invoke() 方法。
6. 通过 Class.newInstance() 和 Constructor.newInstance() 都可以创建类的对象实例，但推荐后者。因为它适应于任何构造方法，而前者只会调用可见的无参数的构造方法。
7. 数组和枚举可以被看成普通的 Class 对待。

通过setAccessible设置为true访问私有变量。

方法的调用是先通过Class类的getMethod获取到Method对象，然后通过invoke调用类，其中前面传递obj实例对象，后面是参数。

通过newInstance获取类的实例对象。

注解

https://www.cnblogs.com/java-chen-hao/p/11024153.html

注解与类 接口 枚举 等都是一个级别的东西，注解主要是用于各种架构里面，方便大家伙使用，最常见的就是ORM数据库，只要用注解就能很方便的使用。

注解的核心就是和反射类似的，有一个注解解释器的角色，根据不同的注解去给他翻译成不同的东西即可。

例如，数据库的table，数据库的cloumn等等。

注解包括生命存活周期，包括生效的位置，注解中定义变量只支持内置变量和注解，可以有默认值，也接受内置变量的数组方式。

示例代码：

```
// 根据不同的注解标签解析生成对应的创建table的sql语句
public static String createTableSql(String className) throws ClassNotFoundException {
    Class<?> cl = Class.forName(className);
    DBTable dbTable = cl.getAnnotation(DBTable.class);
    //如果没有表注解，直接返回
    if(dbTable == null) {
      System.out.println(
              "No DBTable annotations in class " + className);
      return null;
    }
    String tableName = dbTable.name();
    // If the name is empty, use the Class name:
    if(tableName.length() < 1)
      tableName = cl.getName().toUpperCase();
    List<String> columnDefs = new ArrayList<String>();
    //通过Class类API获取到所有成员字段
    for(Field field : cl.getDeclaredFields()) {
      String columnName = null;
      //获取字段上的注解
      Annotation[] anns = field.getDeclaredAnnotations();
      if(anns.length < 1)
        continue; // Not a db table column

      //判断注解类型
      if(anns[0] instanceof DBInteger) {
```

```
        DBInteger sInt = (DBInteger) anns[0];
        //获取字段对应列名称，如果没有就是使用字段名称替代
        if(sInt.name().length() < 1)
          columnName = field.getName().toUpperCase();
        else
          columnName = sInt.name();
        //构建语句
        columnDefs.add(columnName + " INT" +
                getConstraints(sInt.constraint()));
      }
      //判断String类型
      if(anns[0] instanceof DBString) {
        DBString sString = (DBString) anns[0];
        // Use field name if name not specified.
        if(sString.name().length() < 1)
          columnName = field.getName().toUpperCase();
        else
          columnName = sString.name();
        columnDefs.add(columnName + " VARCHAR(" +
                sString.value() + ")" +
                getConstraints(sString.constraint()));
      }


    }
    //数据库表构建语句
    StringBuilder createCommand = new StringBuilder(
            "CREATE TABLE " + tableName + "(");
    for(String columnDef : columnDefs)
      createCommand.append("\n    " + columnDef + ",");

    // Remove trailing comma
    String tableCreate = createCommand.substring(
            0, createCommand.length() - 1) + ");";
    return tableCreate;
  }
```

JMM（Java内存模型）

https://www.infoq.cn/article/java-memory-model-1

https://www.cnblogs.com/czwbig/p/11127124.html

所谓的通信（主存，线程内存，提醒失效）

JMM设计也是不同线程之间的隐式通信，例如有一个变量x，a线程加锁对他加一，b线程获取到这个锁之后读取那么他就是a隐式对这个值的加一操作b知道了，实际上b并不知道谁修改的，所以这个不好说是通信（强行解释那就是匿名信）。

另外，这个对某些对象加锁是保证获取到的线程里在临界代码区对他可以任意修改，当出来临界也就是释放锁，在释放之前会通知主内存这个值无效了，然后把修改后的值写到主存，其他的线程用这个值的时候系统会标记这个值失效，那么他就要再次从主存去获取最新值，这时候就达到了所谓的通信。

另一种通信方式

wait notify notifyall condition

happen-before

主要是为了并发设计一个标准这样，核心是一些规则，最重要的是happen-before规则，

1. 单一线程内a肯定优先后续所有
2. 传递性，a优先b，b优先c，则a优先c
3. 监控锁的释放优先于监控锁的获取（monitor lock，系统级别的锁）
4. volatile 变量规则：对一个 volatile 域的写，happens- before 于任意后续对这个 volatile 域的读。（**这个给忘记了**）

基本的锁都是基于volatile和cas（自旋锁）的基础上实现的，而volatile这些是基于系统的内存屏障由cpu保证的，这些指令大都是不允许指令重排序的，这就保证了顺序。几个内存屏障包括 loadload loadstore storestore storeload，其中load代表从内存读取，store是存储。

缓存一致性

这个性质是系统保证的，单核的话是直接锁定了系统总线，会导致阻塞，多核新的cpu基于MESI（Modified、Exclusive、Share 、Invalid）做的，这四个状态去判断是否要锁定总线，大部分不用，但是不同的线程和核心在切换也是会总是锁定总线的。

- M：被修改的。处于这一状态的数据，只在本CPU中有缓存数据，而其他CPU中没有。同时其状态相对于内存中的值来说，是已经被修改的，且没有更新到内存中。
- E：独占的。处于这一状态的数据，只有在本CPU中有缓存，且其数据没有修改，即与内存中一致。
- S：共享的。处于这一状态的数据在多个CPU中都有缓存，且与内存一致。
- I：无效的。本CPU中的这份缓存已经无效。

顺序一致性模型

https://www.infoq.cn/article/java-memory-model-3

这个是JMM提供的保证性更强的一个特性，他保证一个线程内的程序肯定按顺序执行，正确加锁的在顺序一致性的保证下顺序执行，但是锁内部临界区的内容可以重排序。

volatile

https://www.infoq.cn/article/java-memory-model-4

当读一个 volatile 变量时，JMM 会把该线程对应的本地内存置为无效。

volatile实现：

- 在每个 volatile 写操作的前面插入一个 StoreStore 屏障。
- 在每个 volatile 写操作的后面插入一个 StoreLoad 屏障。
- 在每个 volatile 读操作的后面插入一个 LoadLoad 屏障。
- 在每个 volatile 读操作的后面插入一个 LoadStore 屏障。

volatile特性：

- 可见性。对一个 volatile 变量的读，总是能看到（任意线程）对这个 volatile 变量最后的写入。
- 原子性：对任意单个 volatile 变量的读 / 写具有原子性，但类似于 volatile++ 这种复合操作不具有原子性。

锁

https://www.infoq.cn/article/java-memory-model-5

当线程释放锁时，JMM 会把该线程对应的本地内存中的共享变量刷新到主内存中。

当线程获取锁时，JMM 会把该线程对应的本地内存置为无效。

由于 java 的 CAS 同时具有 volatile 读和 volatile 写的内存语义，因此 Java 线程之间的通信现在有了下面四种方式：

1. A 线程写 volatile 变量，随后 B 线程读这个 volatile 变量。
2. A 线程写 volatile 变量，随后 B 线程用 CAS 更新这个 volatile 变量。
3. A 线程用 CAS 更新一个 volatile 变量，随后 B 线程用 CAS 更新这个 volatile 变量。
4. A 线程用 CAS 更新一个 volatile 变量，随后 B 线程读这个 volatile 变量。

final

写 final 域的重排序规则禁止把 final 域的写重排序到构造函数之外。这个规则的实现包含下面 2 个方面：

- JMM 禁止编译器把 final 域的写重排序到构造函数之外。
- 编译器会在 final 域的写之后，构造函数 return 之前，插入一个 StoreStore 屏障。这个屏障禁止处理器把 final 域的写重排序到构造函数之外

读 final 域的重排序规则如下：

- 在一个线程中，初次读对象引用与初次读该对象包含的 final 域，JMM 禁止处理器重排序这两个操作（注意，这个规则仅仅针对处理器）。编译器会在读 final 域操作的前面插入一个 LoadLoad 屏障。

JMM总结

从 JMM 设计者的角度来说，在设计 JMM 时，需要考虑两个关键因素：

- 程序员对内存模型的使用。程序员希望内存模型易于理解，易于编程。程序员希望基于一个强内存模型来编写代码。
- 编译器和处理器对内存模型的实现。编译器和处理器希望内存模型对它们的束缚越少越好，这样它们就可以做尽可能多的优化来提高性能。编译器和处理器希望实现一个弱内存模型。

Java 程序的内存可见性保证按程序类型可以分为下列三类：

1. 单线程程序。单线程程序不会出现内存可见性问题。编译器，runtime 和处理器会共同确保单线程程序的执行结果与该程序在顺序一致性模型中的执行结果相同。
2. 正确同步的多线程程序。正确同步的多线程程序的执行将具有顺序一致性（程序的执行结果与该程序在顺序一致性内存模型中的执行结果相同）。这是 JMM 关注的重点，JMM 通过限制编译器和处理器的重排序来为程序员提供内存可见性保证。
3. 未同步 / 未正确同步的多线程程序。JMM 为它们提供了最小安全性保障：线程执行时读取到的值，要么是之前某个线程写入的值，要么是默认值（0，null，false）。

小发现（主存数据生效）

例如volatile 和 锁，都有触发时机，例如volatile就是去读数据的时候JMM会让这个本地的值失效，他不得不去主存取值，获取到锁的时候JMM同样也是使线程本地的值失效，这两个都是触发时机，JMM判断他们这些点去置位然后从主存拿最新的值。

Java GC（老年代/新生代/GC Root 不可达）

GC分为几个不同的区域，老年代，新生代，eden区等等，每次扫描的时候会扫描那些通过GC Root不可达的内存，然后进行标记，然后就是整理。这个就是简略的GC步骤了。可以作为GC Root的有静态变量，

> 1.虚拟机栈（栈帧中的本地变量表）中的引用的对象；
>
> 2.方法区中类静态属性引用的对象；
>
> 3.方法区中常量引用的对象；
>
> 4.本地方法栈中JNI（一般说的Native方法）的引用的对象。

从大道理上说，一共有四种基础GC方法。

- Mark Sweep
- Copying Collection
- Mark Compact
- Reference Counting

Android内存抖动/内存泄漏/内存溢出

内存抖动

段时间内有大量对象创建和回收，主要是循环中大量创建回收对象，以及各种buffer的分配回收。

内存泄漏

1. 外部类持有activity的引用
2. 异步执行耗时任务期间，thread timertask等持有activity，在finish的时候实例没有回收
3. Handler内部类造成泄漏，handler为非静态内部类的时候会持有acticity对象，当activity finish的时候handler耗时任务还没有处理完
4. 匿名内部类的使用，各种内部类都会持有对象
5. 使用broadcast receiver/cursor/bitmap时没有及时回收资源
6. 集合的不当使用

内存溢出

outofmemory申请的内存空间超过系统能够分配的。

Android Camera相关

通过context获取到对应的camera service的接口camera manager，然后通过manager获取camera id，打开这个camera的，打开之后在回调里面确定她的分辨率等等，通过自定义button确认是长按的时候开始录制视频，单击就是拍照，其中录制视频是两个surface，一个用于preview一个是用于存储数据，这些控制都是最终的调用camera device，在createCaptureSession的成功回调里开始更新preview同时使用mediarecorder开始录制，录制结束后定制recorder并且reset他，然后通过util去获取视频的某一帧作为界面，在播放的时候是使用的mediaplayer，另外我是通过messager去通知的activity这些信息的，例如录制成功或者拍照成功的路径。这个messager是每次启动这个fragment的时候通过setArgument传递进去的。

Android常用第三方库

room/greendao

都是ORM数据库操作的三方库，都支持通过注解去生成对应的bean或者对应的各种sql语句,其中room是谷歌亲儿子。

okhttp

retrofit

glide

eventbus

jetpack

zhihu luban

Butterknife

LeakCanary（内存泄漏检测）

RecyclerView

lottie动画

Android常用架构

mvc

mvp

mvvm

databinding