[第1章 OS理论基础 3](#_Toc458356499)

[1.1 进程管理 4](#_Toc458356500)

[1.1.1 理论 4](#_Toc458356501)

[1.1.2 Linux进程管理 8](#_Toc458356502)

[~~第2章 Android系统概要~~ 8](#_Toc458356503)

[2.1 静态视图体系 8](#_Toc458356504)

[2.2 源码框架 9](#_Toc458356505)

[2.3 动态体系框架 11](#_Toc458356506)

[2.4 Android应用程序编译 11](#_Toc458356507)

[2.5 Android SDK的体系框架 13](#_Toc458356508)

[2.5.1 标准SDK目录 13](#_Toc458356509)

[2.5.2 SDK的定制化 13](#_Toc458356510)

[2.6 Emulator 14](#_Toc458356511)

[2.6.1 AVD运行 14](#_Toc458356512)

[2.6.2 调试技术 16](#_Toc458356513)

[2.7 工程应用 18](#_Toc458356514)

[2.7.1 Framerwork层的定制化 18](#_Toc458356515)

[2.7.2 添加第三方库 19](#_Toc458356516)

[2.7.3 Android底层移植 21](#_Toc458356517)

[~~第3章 Android系统启动~~ 23](#_Toc458356518)

[~~3.1 Android 启动流程~~ 23](#_Toc458356519)

[3.1 Init进程详解 23](#_Toc458356520)

[~~3.1.1 init.rc文件详解~~ 24](#_Toc458356521)

[~~3.1.2 Action&Service详解~~ 25](#_Toc458356522)

[~~3.1.3 Init进程执行过程~~ 28](#_Toc458356523)

[3.2 Zygote进程详解(重点) 29](#_Toc458356524)

[3.2 .1 ZygoteInit 线程 31](#_Toc458356525)

[3.2 .2 Zygote的分裂-子进程system\_server 32](#_Toc458356526)

[3.2.3 SystemServer线程启动Java层系统服务流程 33](#_Toc458356527)

[3.2.4 Zygote监听应用请求方法-runSelectLoop 36](#_Toc458356528)

[3.2 .5 Home Launch (后续补充) 36](#_Toc458356529)

[3.2.6 实践应用 36](#_Toc458356530)

[3.3 总结 38](#_Toc458356531)

[第4章 Android IPC通信：Binder(重要) 39](#_Toc458356532)

[4.1 Android进程通信模型 39](#_Toc458356533)

[4.2 servicemanager进程 40](#_Toc458356534)

[4.3 Native层服务注册 42](#_Toc458356535)

[4.3.1 MediaServer启动和注册 42](#_Toc458356536)

[4.3.2 mediaserver相关native service的的注册 43](#_Toc458356537)

[4.4 Native层请求服务 44](#_Toc458356538)

[4.4.1 Native客户端请求服务 44](#_Toc458356539)

[4.4.2 Natvie层的C/S通信框架 46](#_Toc458356540)

[4.5 Java层的Binder框架 47](#_Toc458356541)

[4.5.1 Java层服务注册 47](#_Toc458356542)

[4.5.2 Java层客户端请求服务 48](#_Toc458356543)

[4.5.3 Java层Binder框架 49](#_Toc458356544)

[4.5.1 AIDL 50](#_Toc458356545)

[4.6 实践应用 51](#_Toc458356546)

[4.7 总结 51](#_Toc458356547)

[第5章 Android异步的基础-Handler机制 52](#_Toc458356548)

[5.1 原理应用分析 52](#_Toc458356549)

[5.2 Handler使用注意总结 56](#_Toc458356550)

[第6章 APP Framework Layer 56](#_Toc458356551)

[6.1 应用进程管理器-ActivityManagerService 56](#_Toc458356552)

[1，AMS的启动（入口） 57](#_Toc458356553)

[2，startActivity 启动分析（后期分析） 65](#_Toc458356554)

[3，与广播组件的交互 65](#_Toc458356555)

[4，与service组件的交互 65](#_Toc458356556)

[6.2 PackageManagerService 65](#_Toc458356557)

[6.2 WindowManager System 66](#_Toc458356558)

[6.2.1 WindowManager.LayoutParams重要参数分析 67](#_Toc458356559)

[6.2.2 Activity窗体实例分析 70](#_Toc458356560)

[6.3 Android绘图技术-控件系统 72](#_Toc458356561)

[附录A Android虚拟机 74](#_Toc458356562)

[附录B Android JNI框架 74](#_Toc458356563)

[~~附录C Android初始化语言(AIL)~~ 74](#_Toc458356564)

[附录D 双链表 78](#_Toc458356565)

[附录E Android基础类(Native层) 79](#_Toc458356566)

[附录F C++程序设计概要 84](#_Toc458356567)

[1 基础语法部分 84](#_Toc458356568)

[1.1 C与C++与Java基础部分的差异性 84](#_Toc458356569)

[1.2 C与C++中的预编译 85](#_Toc458356570)

[1.3 函数 85](#_Toc458356571)

[2 面向对象 86](#_Toc458356572)

[2.1 面向对象特性：封装，继承，多态性 86](#_Toc458356573)

[2.2 模板 95](#_Toc458356574)

[2.3 异常抛出处理 96](#_Toc458356575)

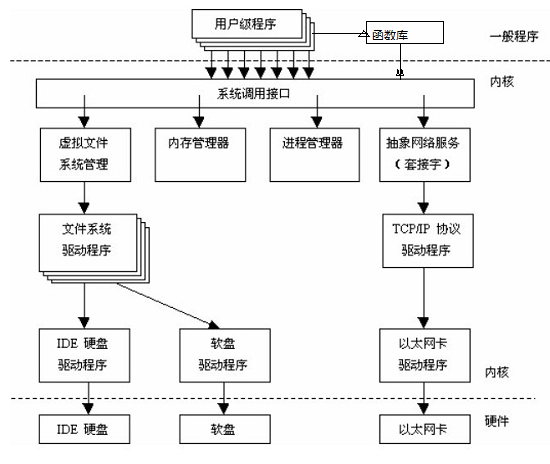
[2.4 STL框架 97](#_Toc458356576)

# 第1章 OS理论基础

操作系统（Operating System，简称OS）是管理和控制计算机硬件与软件资源的计算机程序，是直接运行在“裸机”上的最基本的系统软件，任何其他软件都必须在操作系统的支持下才能运行。对于现代操作系统来说，其基本上会含有如下功能：

* 进程管理（Processing management）
* 内存管理（Memory management）
* 文件系统（File system）
* 网络通讯（Networking）
* 安全机制（Security）
* 用户界面（User interface）
* 驱动程序（Device drivers）
* 中断管理 (Interruption management)

例如，对于Linux系统来讲，其框架如下：



从这里看出，对资源管理控制部分主要是由linux 内核来作用完成，其内核承担了主要的工作部分。这里主要研究进程管理，内存管理。

## 1.1 进程管理

进程管理实质就是对进程做调度，使得多进程能够协同运行，cpu获得充分利用。

### 1.1.1 理论

#### 1.1.1.1 进程概念

进程是正在运行的程序的实例，其有两个主要特点：

第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）。

第二，进程是一个“执行中的程序”，因此其具有周期性。

其含有如下几个特点：

动态性：进程的实质是程序在多道程序系统中的一次执行过程，进程是动态产生， 动态消亡的。

并发性：任何进程都可以同其他进程一起并发执行

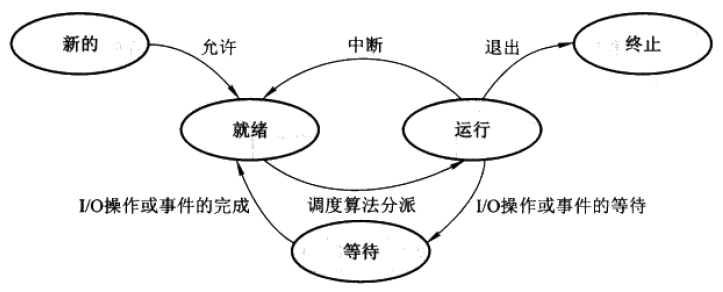
独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位；

异步性：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进

结构特征：进程由程序、数据和进程控制块三部分组成。

#### 1.1.1.2 进程状态

进程状态：新的(创建)，就绪，运行，等待(阻塞),终止



1，进程创建时候，其会对应PCB来表示该进程。

2，进程调度，其实质就是进程在就绪，运行，等待三个状态之间发生变化。

**CPU调度**

目标：cpu利用率最大化

调度准则：cpu利用率，吞吐量，最大化

周转时间，等待时间，响应时间最小化

方式：抢占调度(涉及同步问题)，非抢占方式

调度算法：先来先服务调度(FCFS)，

最短作业优先调度(SJF)，

优先级调度(老化技术来解决低优先级进程的饥饿)，

轮转法调度(RR,时间片轮换调度)

#### 1.1.1.3 进程同步

需要保证进程中的协作中共享数据资源的一致性，所以就要处理进程竞争引起的问题。这里就涉及如下问题：

**1，临界区**

概念：含有共同变量，表或者文件操作的代码段会被多个进程改变。为了保证数据一致性，就要保证同一时刻只有一个进程进入该临界区。其如下：



特性：a，互斥：同一时刻只有一进程进入临界区

b，前进：若临界区内没有进程执行且有进程要进入临界区，那么

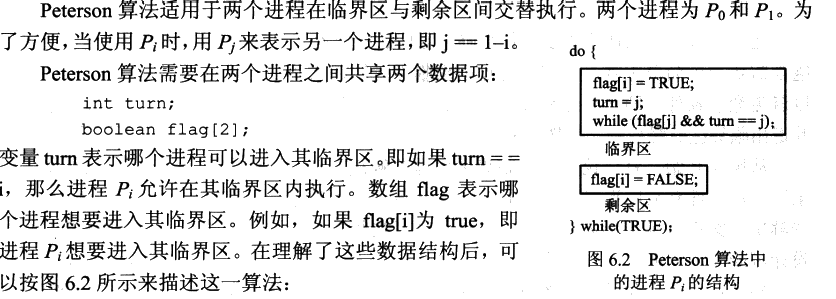
只有不在剩余区执行的进程可以参加选择

c，有限等待

解决方案：

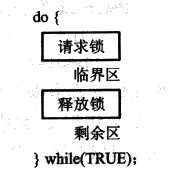
* Peterson算法

Peterson算法是一个实现互斥锁的并发程序设计算法，可以控制两个线程访问一个共享的单用户资源而不发生访问冲突。方法实现：



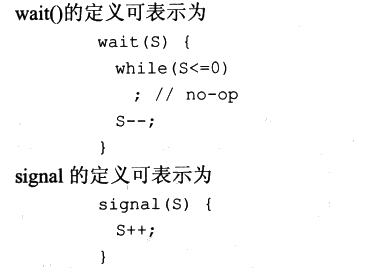
* 锁技术

通过加锁来保证同一时刻只有一个进程可以进入临界区，方法实现如下：

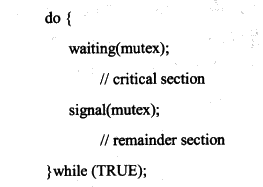


* 信号量

其含有wait与signal两个原子操作，其方法定义如下：



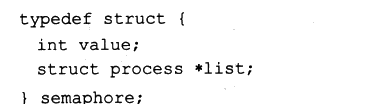
基于方法定义，可以有两种信号量：计数信号量，二进制信号量(互斥锁)。若所有进程共享互斥锁mutex，其实现临界区保护如下：

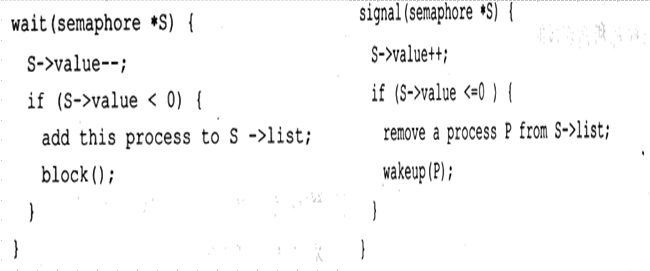


但让使用信号量机制也可以实现不同进程之间某些代码执行先后顺序的同步。

缺点：wait与signal的实现方式会导致进程处于**忙等待**状态。其使得所有进程都一直处于运行-就绪状态，其占用了cpu的时钟。若占用时间较短时候，是可以使用该方式执行(称**自旋锁**)，但是若时间较长的时候，浪费了cpu的资源。可以采用其他实现方式，将等待的进程放入等待队列，通过wakeup唤醒到就绪队列在执行。

**改进型的wait,signal方法定义如下**：





**2，经典同步问题**

有限缓冲(生成者与消费者)，读-写问题，哲学家就餐问题

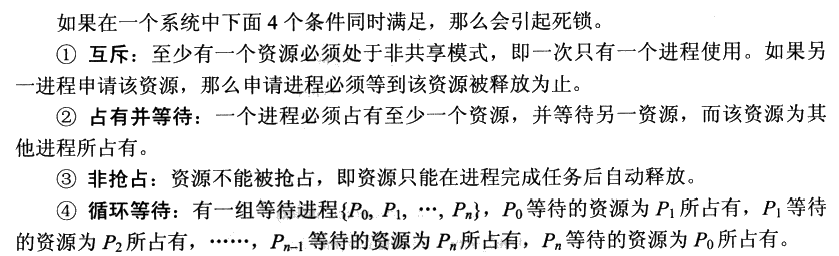
**3，总结**

信号量机制很好的处理了进程同步问题，但是其也会带来新的问题：**死锁和饥饿**，所以怎么避免死锁类问题也将是进程同步中比较重要的一个议题。

#### 1.1.1.4 死锁

**1，概念**

一组进程中每个进程都在等待同一个事件，而该事件确由该组进程中另外一个进程引发，那么该组进程就进入了死锁状态。 死锁其具有如下几种特点：



比较常见的AB-BA的死锁。

**2，死锁的处理方式**

死锁的处理常用的有两种方式：死锁预防和死锁避免

**2.1，死预锁防**

**2.2， 死锁避免**

资源分配图算法，银行家算法(安全检测算法&资源请求算法[回溯])

### 1.1.2 Linux进程管理

# ~~第2章 Android系统概要~~

目前，android与ios作为移动开发的两个主流平台，其的应用价值极其广阔。要想深入学习android，必须先理解其体系结构。本章节主要介绍其体系结构(Android4.4.2)，且分如下几个小结：

1，静态视图体系

2，编译体系

3，AVD的运行

## 2.1 静态视图体系

Android系统采用分层次的结构(如下图)，



其从上往下可以分为如下四个层次：

1）应用层：应用程序(基于SDK开发对应就是apk)，但基于NDK开发(实质是为了运行效率或者代码安全考虑，采用JNI的方式给Java层调用，特别在智能HW开发，为了直接操作HW功能)

2）应用框架层：其由 Java/Jni两层组成。其中，对SDK开发应用来讲：

Java 层为应用层开发提供了API，而Jni层是为了调用系

统运行库设计。两者的结合提供了应用层开发的完整机

制支持。 对于NDK开发来说，JNI层为其提供API接口。

3）Android的运行环境和系统运行库层：该部分都是与机器有关系的，

其分为两个部分组成：

3.1)系统运行库，主要支撑JNI的本地服务(JNI调用)，由

**各种系统运行库(google编写)或者第三方库(引入)**

组成。

3.2)Android运行环境，简而言之就是为Apk提供一个

运行环境(Android5.0 ART发生本质变化)。

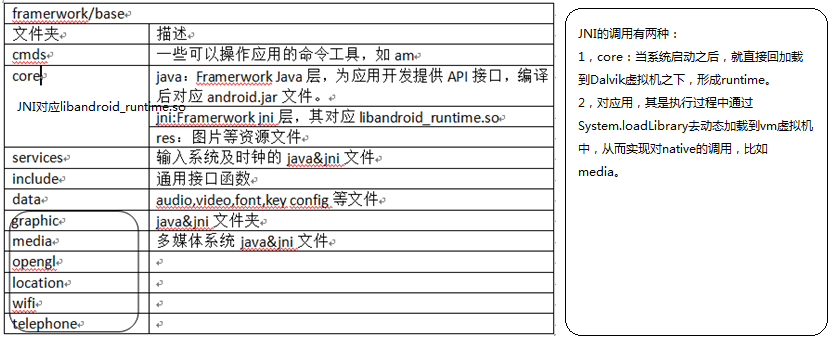
4）Linux内核层:其对linux内核做了裁剪，且增加新的Android驱动。

## 2.2 源码框架

上面一小节说明了静态视图的框架，那么本节来研究主要部分的源码的结构分布图

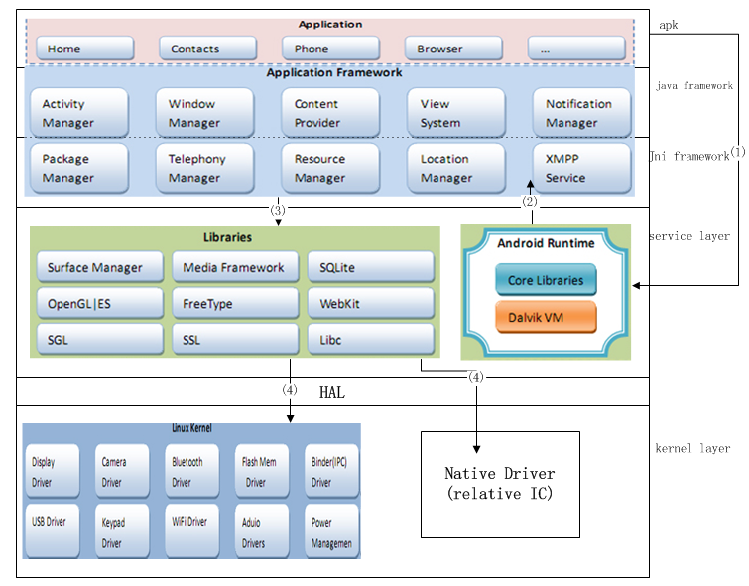


其中，



## 2.3 动态体系框架

该小结主要讨论一问题：当系统运行起来之后，应用程序是怎么在系统上工作的(基于SDK开发)。如下图：



|  |
| --- |
| （1） Apk运行在虚拟机 |
| （2）虚拟机执行代码过程调用JNI接口函数。 |
| （3）JNI 函数调用本地系统库 |
| （4）本地库调用驱动程序来驱动硬件 |

从上面的运行过程看，对Android系统深度开发来说，其可以有如下工程应用：

1，Framerwork层的定制化，包含SDK的定制化。

2，底层运行库中，引入第三方自定义库，以支持额外功能。

3，为支持HW，底层的移植技术

4，引入新的调试工具

后面章节会专门说明这几种应用。

## 2.4 Android应用程序编译

应用程序的编译有三种方式执行：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1，命令行自动编译，其流程如下图：       |  | | --- | | 使用aapt生成R.java类文件  使用android SDK提供的aidl工具把.aidl转成.java文件 | | 编译.java类文件生成class文件，使用android SDK提供的dx生成classes.dex文件 | | 生成未签名的apk安装文件 | | 使用jdk的jarsigner对未签名的包进行apk签名，调试测试应用程序，而后发布 | |
| 2，ant自动编译打包：  定义build.xml脚本，暂时不详细介绍 |
| 3，Eclipse+ADT组合方式：  **编译：**  1) 编译当前项目，并将文件转换为dex。  2) dex file + resource files ->package为Android 包(apk)。  **运行:**  1）启动创建AVD虚拟设备  2）将编译的apk包推送到AVD设备  3）启动应用程序   |  | | --- | | **Android SDK**  SDK：（software development kit）软件开发工具包。被软件开发工程师用于为特定的软件包、软件框架、硬件平台、操作系统等建立应用软件的开发工具的集合,为开发者提供了库文件以及其他开发所用到的工具  **Android ADT**  ADT是英文 Android Development Tools 的简称，ADT是Eclipse中构建Android应用程序IDE插件，ADT扩展了Eclipse的功能，让您快速建立新的Android项目、创建应用程序的用户界面。ADT可以帮助Eclipse找到Android sdk，再加上Android框架的API组件调试使用Android SDK工具，以便APK分发给应用程序。 | | 从上面得出：真正操作编译，运行调试等过程的为**ADT指向的SDK包中的工具。** | |

## 2.5 Android SDK的体系框架

主流的应用开发是基于Android SDK提供的API来完成的，所以深入了解SDK是非常重要。

### 2.5.1 标准SDK目录

其结构目录如下：

|  |  |
| --- | --- |
| **目录名字** | **用途说明** |
| add-ons | 存着附加库(非android部分)，比如GoogleMaps等 |
| **platforms** | 每个平台的SDK真正的文件，保存不同android版本的一些信息和API函数，其中对于framerwork层对应 android.jar |
| platform-tools | 调试目标设备的工具，如adb等。 |
| samples | samples是**Android SDK**自带的默认示例工程 |
| **tools** | 作为SDK根目录下的tools文件夹，这里包含了重要的运行，管理和调试目标设备的工具，比如android，emulator，ddms，monkeyrunner等。 |
| 其他目录 | docs(API说明文档)，market\_licensing |

### 2.5.2 SDK的定制化

一般来说，在TV行业中，公司都会自己对android系统增添一些功能模块。所以，因为功能的扩展部分，就不能使用google官方的sdk包。那么，怎么定制SDK包？

|  |
| --- |
| 1，Android源码根目录下编译：make PRODUCT-sdk-sdk (注：若API发生修改，先执行make update-api ) |
| 2，编译介绍后，其会out目录：  out/host/linux-x86/sdk：      out/target/product/generic：    其中：android-sdk\_eng.mtk70982\_linux-x86 即为定制化的sdk，基于该sdk包就可以做应用层的开发。 |

## 2.6 Emulator

### 2.6.1 AVD运行

当一系统编译出来SDK之后，其中含有tools工具，而且还含有img，就可以考虑系统的启动。

|  |
| --- |
| vi ~/.bashrc |
|  |

**1，android命令：**

|  |
| --- |
| **android：** |
| **android list target ：**目标设备信息： |
| **android create avd -n [name] -t [target id]** :基于target id目标设备创建一虚拟设备  ；其流程：  1，当创建成功之后，其会~/根目录之下产生.android文件夹：    2，avd目录之下创建name.avd文件夹目录，其中关键是配置config.ini: |
| **android list avd：**列出avd设备信息 |
| **android -h :**可以列出其详细用法 |

**2，emulator命令：**

|  |
| --- |
| emulator @bob  **[-logcat d]**：启动avd，实质就是读avd对应的config.ini，而后虚拟HW设备上启动器其对应的img文档 ，若启动中就要答应log，就加入logcat命令。 |
|  |
| **emulator -help** ：可以获取emulator的大量使用信息 |

### 2.6.2 调试技术

#### ADB(Android Debug Bridge)

ADB是一个 客户端-服务器端 程序, 其中客户端是你用来操作的电脑, 服务器端是android设备。要保证adb是可以使用的，必须保证adb连接到目标android设备。其常用命令如下：

|  |  |
| --- | --- |
| 类别 | 命令 |
| 联通 | connect <host>[:<port>]  - connect to a device via TCP/IP，Port 5555 is used by default if no port number is specified. |
| disconnect [<host>[:<port>]]  - disconnect from a TCP/IP device， Port 5555 is used by default if no port number is specified. |
| 环境切换 | **adb shell**  - run remote shell interactively,enter linux  environment of device  adb shell <command> - run remote shell command  adb emu <command> -run emulator console command  adb logcat [ <filter-spec> ] - View device log |
| adb bugreport  - return all information from the device， that should be included in a bug report. |
| 操作设备 | adb devices [-l] - list all connected devices  adb push <local> <remote> - copy file/dir to device  adb pull <remote> [<local>] - copy file/dir from device  adb sync [ <directory> ] - copy host->device only if changed |
| adb install [-l] [-r] [-s] [--algo <algorithm name> --key <hex-encoded key> --iv <hex-encoded iv>] <file>    - push this package file to the device and install it,  ('-l' means forward-lock the app)  ('-r' means reinstall the app, keeping its data)  ('-s' means install on SD card instead of internal storage)  ('--algo', '--key', and '--iv' mean the file is encrypted already)    adb uninstall [-k] <package>  - remove this app package from the device ('-k' means keep the data and cache directories) |
| adb help ：帮助说明 | |

#### DDMS(Dalvik Debug Monitor Service)

Dalvik Debug Monitor(对应用做性能分析),一般集成与eclipse中。

#### Eclipse中程序调试

Eclipse中常见的是对Java程序做调试功能。这里分为两种类型：

1) 运行APP

2） Framework Java层的框架。

|  |
| --- |
| APP对应代码调试，模拟器运行起来之后，通过设定对应code断点做调试，后期再整理调试技术。 |
| Framework Java层的调试是需要对源代码整合出来不同文件，基于这些文件做调试：  1，SDK包->make PRODUCT -sdk-sdk-编译  2，ADT包->a,安装eclipse软件，其中同时安装Eclipse CDT插件  b,安装javaee 64bit for linux.设定路径参数：  export ECLIPSE\_HOME=你的eclipse路径  export PATH =$PATH:$ECLIPSE\_HOMe  c,进入sdk/clipse/scripts目录执行：  build\_server.sh **~/adt**[存放ADT的数据包]  d，运行Ubuntu下的Eclipse软件，导入ADT与SDK包，Android  开发环境架设完毕。  3，调试步骤  参考：http://www.360doc.com/content/12/1227/20/9462341\_256653851.shtml  **Note：**再次编译code时，一定要删除eclipse编译生成bin文件 |

## 2.7 工程应用

### 2.7.1 Framerwork层的定制化

Framerwork的定制化可以分为两种情况：

1，基于Framerwork的框架，修改该框架代码，使其满足某种需求特性。

**专业需求：**对Framerwork层了解深刻，后期框架研究部分做介绍。

2，增加新的功能组件，为独立功能支持部分，为应用开发提供新增API。

**专业需求：**对整合方式很了解。

|  |  |
| --- | --- |
| **1，Java Framework层：**  **build/core/pathmap.mk**: FRAMEWORKS\_ BASE\_ SUBDIRS  and for \*.aidl, link it in following path:  **frameworks\base\Android.mk**: LOCAL\_SRC\_FILES   |  | | --- | | 注意，当SDK新增加接口，有两种存在形式：  1)定义为@hide，即不对外开放  2)对外开放，注意若开放接口调用@hide接口要处理，且执行make update-api(方式比较复杂) | |
| **2，Jni Framework层：**   |  | | --- | | 1，注册runtime运行环境，android系统启动时加载。  **path:** \base\core\jni  **files:**  added files ,AndroidRuntime.cpp(gRegJNI添加注册)  **参考：**register\_android\_media\_AudioRecord | | 2，应用程序运行时候，调用System.loadLibrary动态加载。  参考例子：media(framerwork层代码都存在同一目录下)  **path：**frameworks\base\media\jni(media\_jni)  **弊端：**资源开销比较大 | |
| Note:查找find .|xargs grep -ri "LOCAL\_SDK\_VERSION" -l |

### 2.7.2 添加第三方库

#### 2.7.2.1 Android源代码中集成

第三方库可以分为两种：1，编译代码生成 2，已经存在binary文件

|  |  |  |
| --- | --- | --- |
| **NOTE**  1，指定LOCAL\_MODULE\_PATH，其就会存放到指定目录之下。  2，如果没有指定，就会存放到默认目录之下，记住pre-build只是copy动作。  out/target/product/generic/obj/STATIC\_LIBRARIES  out/target/product/generic/obj/EXECUTABLES  out/target/product/generic/obj/SHARED\_LIBRARIES  out/target/common/obj/JAVA\_LIBRARIES  (out\target\product\generic\obj\\*对应  out\target\product\generic\system\\*) | | |
| 类别 | 库类型 | 方法 |
| **build**  **source**  **file** | execuable | 【init】进程  #源代码路径：system/core/init  #指定其输出路径(Android.mk)：  **LOCAL\_MODULE\_PATH** := $(TARGET\_ROOT\_OUT)  LOCAL\_UNSTRIPPED\_PATH := $(TARGET\_ROOT\_OUT\_UNSTRIPPED)  include $(BUILD\_EXECUTABLE)  #编译为执行文件，输出目录：  out\target\product\generic\root  out\target\product\generic\symbols |
| so lib | 【libmedia\_jni.so】  LOCAL\_MODULE:= libmedia\_jni  include $(BUILD\_SHARED\_LIBRARY)  输出：  #out/target/product/generic/obj/SHARED\_LIBRARIES/libmedia\_jni\_intermediates/LINKED/libmedia\_jni.so  #out/target/product/generic/system/lib/libmedia\_jni.so  #out/target/product/generic/obj/lib/libmedia\_jni.so  #out/target/product/generic/symbols/system/lib/libmedia\_jni.so  $(TARGET\_OUT\_SHARED\_LIBRARIES)->out/target/product/generic/system/lib |
| a lib | include $(BUILD\_STATIC\_LIBRARY) |
| **pre-**  **build**  **binary** | execuable | 【am】进程  include $(CLEAR\_VARS)  LOCAL\_MODULE := am  LOCAL\_SRC\_FILES := am  LOCAL\_MODULE\_CLASS := EXECUTABLES  LOCAL\_MODULE\_TAGS := optional  include $(BUILD\_PREBUILT)  #输出目录：  out/target/product/generic/obj/EXECUTABLES/  out/target/product/generic/system/bin/ |
| so lib | LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS)  LOCAL\_PREBUILT\_LIBS :=  include $(BUILD\_MULTI\_PREBUILT) |
| a lib | LOCAL\_PATH := $(call my-dir)  include $(CLEAR\_VARS)  LOCAL\_PREBUILT\_LIBS :=  LOCAL\_STATIC\_LIBRARIES :=  include $(BUILD\_MULTI\_PREBUILT) |
| **从上面分析可以看出：Android对编译生成文件输出目录做了统一的管理。所以，对系统工程师来说，只需要按照其规则写出来正确的Android.mk文件，其就输出到对应的文件夹中。** | | |

#### 2.7.2.2 Eclipse中添加

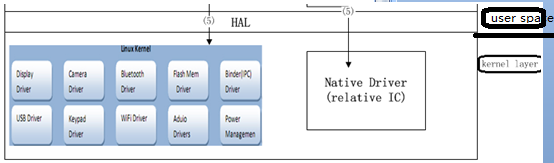
应用开发过程中，要使用第三方的库，可以在Eclipse导入，参考文章：

http://blog.csdn.net/thl789/article/details/7918093。

### 2.7.3 Android底层移植

#### 2.7.3.1 移植技术概要

从静态图查看，即将如下部分联通到系统中去：



从层次架构抽象来考虑分三个部分：

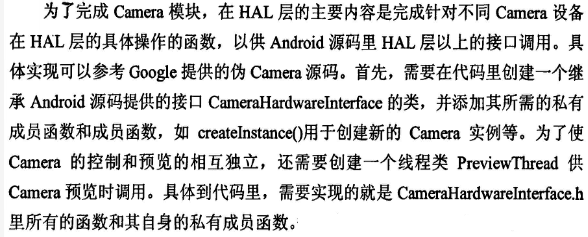
|  |  |  |
| --- | --- | --- |
| **1，Kernel部分：**  驱动程序整合到系统中,，有如下两种方式：   |  | | --- | | 1，驱动直接编译到kernel部分。 | | 2，驱动编译为module，动态插入系统。(一般native ic driver) | |
| **2，kernel部分驱动与userspace 通信[AL]：**  对该驱动创建一设备节点，通过Ioctl方式来不同空间间通信。  **参考：**http://blog.csdn.net/w501271370/article/details/22946805 |
| **3, AL层与native Libraries对接：** |

因一般Driver都已经Ready，所以工作量主要就是在第三步，其存在两种方式：

|  |
| --- |
| 方式1：User Adaption Layer为HAL，已经定义了逻辑，需要填入Driver Abstract Layer层代码来实现。 |
| 方式2：根据上层库的调用，协商定义一套通用API。(比如MTK Dmx驱动使用就是采用方式2) |

#### 2.7.3.2 实例：Camera移植

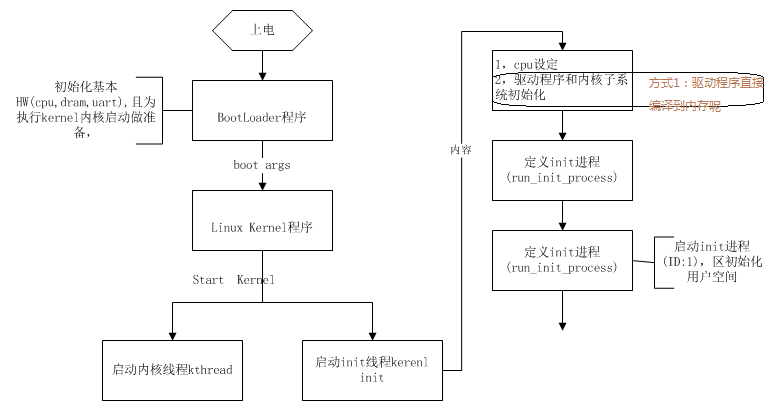
因Android有Camera HAL层，故采用方式1的实现。



# ~~第3章 Android系统启动~~

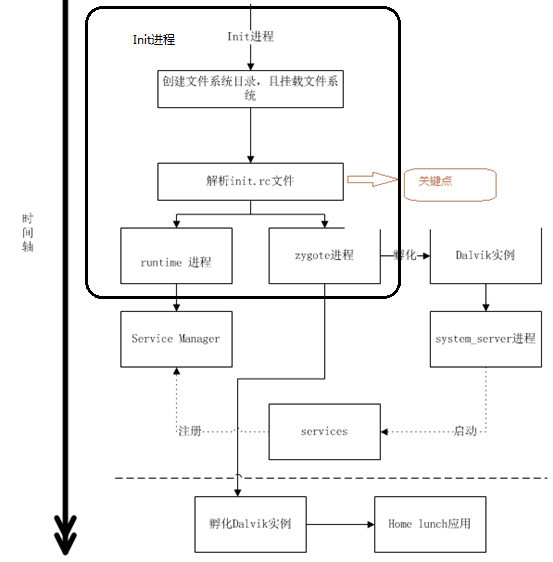
## ~~3.1 Android 启动流程~~

Android启动模式分为normal mode,safe mode ,recovery mode ,factory mode,fastboot mode。那这里只是研究正常启动流程，如下框架：



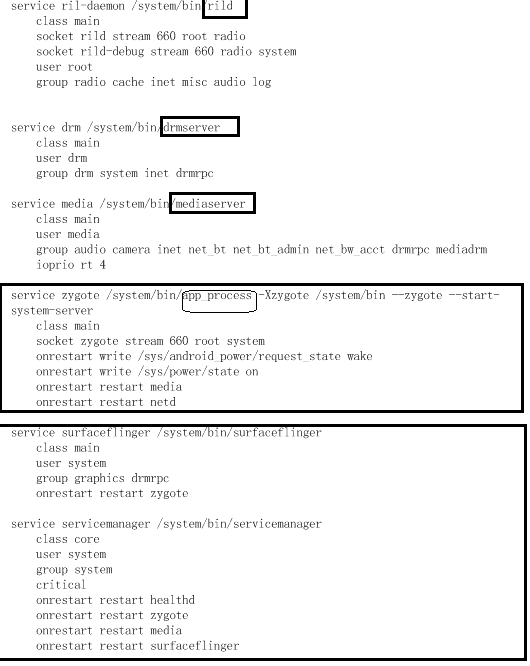
## 3.1 Init进程详解

其源代码存在system\core\init，其大概执行过程如下：



### ~~3.1.1 init.rc文件详解~~

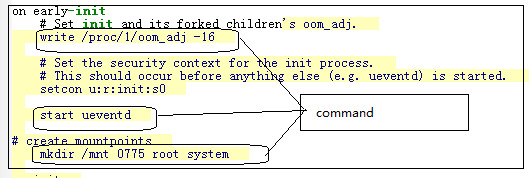
通过init.rc脚本去启动进程，该地方就是存在一扩展：启动脚本下动态加载模块(insmod)。init.rc(system/core/rootdir)的文件格式是按照Android初始化语言(AIL)定义(见**附录C**)。



### ~~3.1.2 Action&Service详解~~

**1, Action**

如Action例子，如下：



actioin实际是去执行对应command，其command执行是通过调用对应的函数，如下：

|  |
| --- |
|  |
| 函数实体定义:system\core\init\keywords.h |
| 对于xx\_start，其最终调用的是service\_start函数(system\core\init\init.c),主体代码如下：    综合得出，其主要做了两件事情：  1，启动service，为该service创建一对应进程，  2，且为该service创建socket套字节，便于进程间的通信。  **所以，Action Command xx\_start命令的实质就是启动Service(进程)。** |

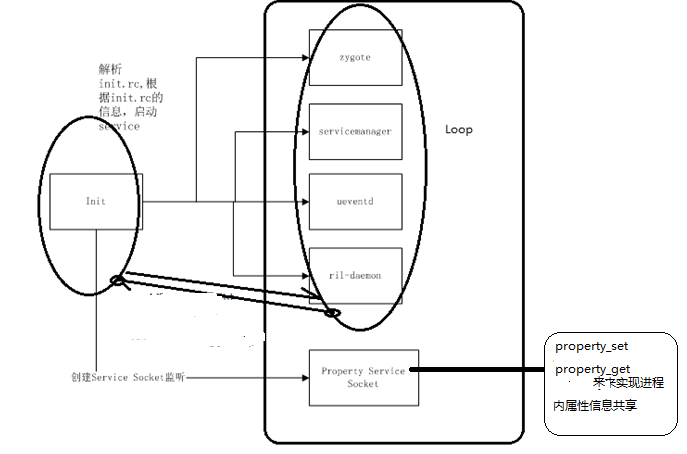
**2, Service**

Service其实没有显示启动它，实质都是通过Service与command关联，通过启动Command来将Service以进程方式来启动。

|  |
| --- |
|  |

### ~~3.1.3 Init进程执行过程~~

通过main函数分析，其执行流程如下：



Init进程实质上：

1，创建了Service(实质就是进程)，且为每个service创建了socket(TCP)，可以实现进程通信。

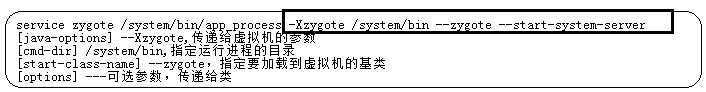
2，Init中通过一Loop来管理所有的进程状态，且通过server socket来接受到一些属性信息的变化情况。且属性区可以理解为一公共区间，可以通过get\_property来获取。

## 3.2 Zygote进程详解(重点)

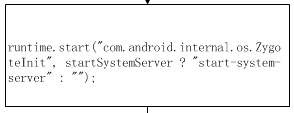
Zygoe进程为守护进程，其Android正常工作的基础。若守护进程死掉了，那么系统肯定挂掉了。Init进程启动了Zygote进程，之后整个流程框架图如下：

|  |
| --- |
|  |
|  |

那么Zygote进程其执行何种行为是本章研究重点，其main函数位于：frameworks\base\cmds\app\_process\app\_main.cpp。



传递了四个参数给main函数，执行流程最核心函数为：



|  |
| --- |
| 研究runtime.start启动过程执行过程如下： |
| 实质是执行framerwork\base\core\jni\AndroidRuntim.cpp : AndroidRuntime:start方法：  1） 创建VM虚拟机：AndroidRuntime::startVm  2） 注册JNI函数：startReg  3）进入Java世界：通过JNI方式调用ZygoteInit.main()方法 |

### 3.2 .1 ZygoteInit 线程

执行ZygoteInit.main()创建了Java的ZygoteInit线程，其在如下文件夹之下：

\frameworks\base\core\java\com\android\internal\os\ZygoteInit.java

其主要有如下几个部分的工作：

|  |  |  |
| --- | --- | --- |
| 1）registerZygoteSocket(); 为zygote进程创建一ServerSocket，使得ActivityManagerService便可以通过此socket来让zygote启动新的应用程序。   |  | | --- | | 基于linux的c的socket来实现，而并非基于Java的标准库socket机制。其对象结构如下： | | sServerSocket = new LocalServerSocket(  createFileDescriptor(fileDesc)); | |
| 2）preload(); 预加载class和resource资源，可以保证不同应用共享框架层资源，  提高运行效率。 |
| 3）startSystemServer();启动system\_server进程，来开启Android中的Native System Service和Java层的System Service，这是应用层框架运行的基础。 |
| 4）执行MethodAndArgsCaller的run方法，实际是执行SystemServer.main. |
| 5）runSelectLoop(); 处理新应用程序的运行请求 |

所以，从这里可以得出ZygoteInit主要功能是：

1，Zygote与其他程序的通信方式：socket通信。

2，启动System Service进程[native]，**且该过程使用了JNI函数与Java层交互机制的技术。**

### 3.2 .2 Zygote的分裂-子进程system\_server

system\_server进程创建是执行ZygoteInit线程中startSystemServer()方法：调用Zygote.forkSystemServer创建system\_server子进程，且子进程中执行

handleSystemServerProcess(parsedArgs)。这个时刻就是Zygote分裂出了system\_server进程（启动Android系统的服务）：

1，fork特性：父进程和子进程共享在执行fork之前的所有资源，即子进程含有对父进程fork时间点之前所有资源的一份拷贝。

2，重点分析handleSystemServerProcess，如下：

|  |
| --- |
|  |
| system\_server进程创建了SystermServer线程(有Looper)，执行SystemServer.main方法如下： |

### 3.2.3 SystemServer线程启动Java层系统服务流程

线程执行如下代码：

|  |
| --- |
| ServerThread thr = new ServerThread();  thr.initAndLoop(); |
| 其主要功能：  1，创建了Java层系统服务对象，且注册服务到ServiceManager  2，建立系统服务对象关联关系  3，启动服务 |

重点介绍ActivityManagerService，PackageManagerService，WindowManagerService，PowerManagerService 的工作过程，如下：

|  |
| --- |
| 功能1,2部分对应代码(执行图)：    要搞清如下两个关键点：  1，所有的系统服务都只创建一个实例，即为单例实例。  2，Context是一个接口，通过它可以获取并操作Application对应的资源，类，甚至包含Application中的四大组件，其具体实现类为ContextImpl，通过init方法来不断向ContextImpl中添加关联资源。 |
| 功能3部分对应代码：   |  | | --- | | wm.displayReady();  pm.performBootDexOpt();  ActivityManagerNative.getDefault().showBootMessage(  context.getResources().getText(  com.android.internal.R.string.android\_upgrading\_starting\_apps),  false);  wm.systemReady();  power.systemReady(twilight, dreamy);  pm.systemReady();  display.systemReady(safeMode, onlyCore);  ActivityManagerService.self().systemReady(Runable) | |

#### 1服务创建的关键函数的分析

|  |  |
| --- | --- |
| Context context = ActivityManagerService.main(factoryTest); | 1，SystemServer线程中创建了ActivityManagerService对象，实质就是Java应用进程管理器。  2，创建Context对象，它背后所指向的Application环境与framework-res.apk有关，此时系统Context中只有LoadedApk（未绑定ApplicationInfo）对象。  3，为system\_server进程创建一个Android运行环境 |
| 参考：[Part1：main函数](#_Part1：main函数) | |
| pm= PackageManagerService.main(context, installer,factoryTest!= SystemServer.FACTORY\_TEST\_OFF,  onlyCore); | SystemServer线程  创建PackageManagerService对象来对包生命周期和组件间通信管理。 |
| 参考：[6.2 PackageManagerService](#_6.2__PackageManagerService) | |
| ActivityManagerService.setSystemProcess(); | 创建一代表system\_server的ProcessRecord，交给AMS做统一管理。 |
| 参考：[Part2: AMS的setSystemProcess函数](#_Part2:__AMS的setSystemProcess函数) | |
| wm= WindowManagerService.main(context, power, display, inputManager,wmHandler, factoryTes!= SystemServer.FACTORY\_TEST\_LOW\_LEVEL,  !firstBoot, onlyCore); | 创建WindowManagerService实例 |
| 参考：[6.2 WindowManager System](#_6.2_WindowManager_System) | |
| 注意：如下public static void main(String[] args)函数入口就表示执行Java新线程 | |

#### 2 服务启动关键函数分析

|  |  |
| --- | --- |
| ActivityManagerService.self().systemReady(Runable) | 启动Launch进程 |
| 参考：[Part3：AMS中的systemReady函数](#_Part3：AMS中的systemReady函数) | |

### 3.2.4 Zygote监听应用请求方法-runSelectLoop

当Zygote分裂之后，其中两个进程后续功能：

1) system\_server进程开启Android系统服务，用来专门处理对服务的请求。

2) Zygote来监听应用请求，但存在请求创建VM实例，实际就是应用进程。

(创建方式也是fork)

### 3.2 .5 Home Launch (后续补充)

请参考：[Part3：AMS中的systemReady函数](#_Part3：AMS中的systemReady函数)

HomeLaunch启动过程，实际也可以看成是一普通Launch应用的启动，其

可以分为如下几个步奏：

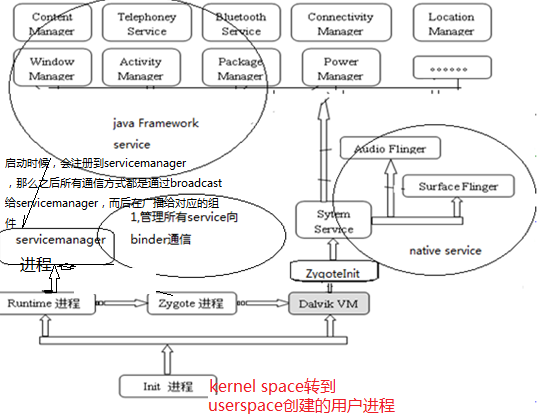
|  |
| --- |
| 1，-----入口？  2，调用    创建应用进程（实质Zygote进程 fork的子进程），而后执行ActivityThread.main  创建主线程，及为UI线程(含Looper)。  3，基于Handler机制来实现该应用进程子线程与主线程的通信。Handler机制本质实现同进程子线程的通信。 |

### 3.2.6 实践应用

|  |
| --- |
| **案例1:[Mediatek]linux中的关于TV部分的middleware的api移植到Android上面。**  分析：因该套api非常稳定，不能说在android TV上重新实现一套，所以采用JNI技术来移植该套api，其中JNI注册采用启动时候注册，其TV部分在Android上的框架变为： |
| **案例3：开机速度优化**  bootchart应用 |
| **案例4：定义启动自己的launch界面**  **实际上不需要修改Framework层任何东西，真正要做的就是开发Launch App** |
| **案例5：系统启动过程注册JNI函数**    Java应用开发过程中，也可以使用JNI开发，实质：  JNI/C代码使用NDK工具编程生成so文件库，  Java文件中加载该库使用。  关键点：Java Native函数与JNI函数命名规则必须满足，这样子才能导致注册到VM的能够找到自己的函数。 |
| **案例6：删除掉不适用的系统应用** |

## 3.3 总结

从启动过程来看，其流程可以总体概述为如下：



且其使用了技术：

1，native创建父子进程的通信方式：socket套字节

2，JNI与Java层的交互技术(Java反射与JNI编程技术)

# 第4章 Android IPC通信：Binder(重要)

如果该章节没有办法明白，后续的源代码就没有办法继续深入理解，所以该章节为非常重要部分。其中，主要涉及设计思路：

1） 代理技术

2） C/S的通信机制

再看本章节时候，请先看[附录E Android基础类(Native层)](#_附录E__Android基础类(Native层))

## 4.1 Android进程通信模型

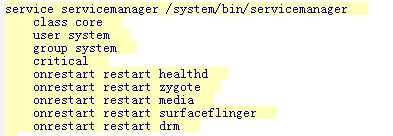
Android系统的通信机制是基于C/S来实现，将包含service服务的进程成为Server进程，将调用service服务的进程称为Client进程，其通信框架如下：

|  |
| --- |
| 整体框架部分有三个块组成：  1，client(客户端)  2，ServiceManager(Android的一个全局service，也是总管所有server的进程，在init.rc中启动)  3，server(服务)  其三种之间的关系如下： |
| （1）注册服务  1）获得ServiceManagerService服务的代理  2）注册实际服务registerService(servicename,ActivityManagerService{真实服务对象})  3）查询服务getService(servicename)，实际返回服务的一代理。 |
| （2）查询服务  实质就是查询ServiceManagerService维护的一service list，找到对应目标服务。 |
| （3）客户端使用服务的流程：调用代理服务业务处理方法 |

## 4.2 servicemanager进程

从上面的通信三步来看，servicemanager是所有service的总管，当客户端需要使用服务时候，必须通过servicemanager来查询到该service，而后在使用它。所以，理解servicemanager的整个过程就非常重要。这里说的是ServiceManager服务的一个实体。

Init进程创建ServiceManger，其init.rc中对应的脚本语句：



执行servicemanager,其主体函数位frameworks/native/cmds/servicemanager.



主要执行了如下两个动作：

|  |
| --- |
| 且确定该进程就是系统的serverManager管理进程。 |
| **[监听]binder\_loop(bs,svcmgr\_handler);**   |  | | --- | | 循环监听，将binder中获取的数据丢给svcmgr\_handler处理。 | | 1，SVC\_MGR\_ADD\_SERVICE：注册服务  **哪些service是可以注册服务(可以定制化部分)：**  root用户，system用户及如下：    这些都是servicemanager启动时刻预加载的服务。 | | 2，SVC\_MGR\_GET\_SERVIC: SVC\_MGR\_CHECK\_SERVICE获取服务  3，SVC\_MGR\_LIST\_SERVICES 得到当前系统已经注册的所有服务的名字 |   servicemanager作为守护进程，实质就是动态维护servicelist列表，维护<servicename,Bp,Bn>的关系。类似关系： |

## 4.3 Native层服务注册

### 4.3.1 MediaServer启动和注册

以MediaServer(音视频服务总管，其作为一个进程，init.rc中启动)为研究体，来研究该对象注册到serviceManager和启动的整个过程[也是一个CS模型]。

|  |
| --- |
| MediaServer入口：frameworks/av/media/mediaserver/ main\_mediaserver.cpp,其  中定义关于如相关性的服务 |
| MediaServer中含有的一些service： |
| 从这里看出，Android中服务的注册框架图如下： |

### 4.3.2 mediaserver相关native service的的注册

|  |
| --- |
| mediaserver  执行函数入口为 framerwork/av/media/mediaserver/main\_mediaserver.cpp |
| mediaserver中创建了一子进程来注册服务到servicemanager，其中步奏如下：   |  | | --- | | 1，sp<ProcessState> proc(ProcessState::self());  获取该进程的ProcessState对象，每个进程只有唯一的ProcessState对象。 | | 2，sp<IServiceManager> sm = defaultServiceManager();  请先参考附录G：IPC通信基础类，这个函数主要实现两个功能：  1，创建BpBindre对象，其中handler=0，指向ServerManger的BBinder，其transact方法中建立了与硬件Binder设备的直接通信。  2，基于BpBinder对象创建BpServiceManager对象，实现IserviceManager的对象的所有业务函数，这些业务中调用了BpBinder中transact通信接口。  所以，实质就是获取到ServiceManager的代理，通过该代理来管理service。 | | 3，注册服务到servicemanager，其通过获取defaultServiceManager获取到serviceManager的代理，而后调用addService方法来注册  AudioFlinger::**instantiate**();  MediaPlayerService::**instantiate**();  CameraService::**instantiate**();  AudioPolicyService::**instantiate**();  registerExtensions();[可以自定义扩展，暂时还未定义] | | 4，  1）启动一新线程新线程，告诉binder BC\_ENTER\_LOOPER  ProcessState::**self**()->startThreadPool();  2)该进程所在主线程，告诉BC\_REGISTER\_LOOPER客户端请求服务详解(服务的客户端)  IPCThreadState::**self**()->joinThreadPool();  实质上就是与talkWithDriver。 | |
| 从整个过程中可以看出，进程中有两个线程在工作，那这个地方是否可以根据服务数量的多少动态的调整线程数量？ |

## 4.4 Native层请求服务

这里以AudioFlinger为例子，讨论其客户端申请该服务的过程，其也满足C/S的基本模型：

1，向ServiceManager查询该服务，放回BpBinder。

2，通过该Binder对象去创建对应的BpxxxService，这样子就获取了该Service

的代理

3，客户端就可以向该Service代理发送请求。

4，Service代理转发给Binder驱动

5， 真正的Service(服务端)会从Binder驱动拿到请求之后，将其做处理。

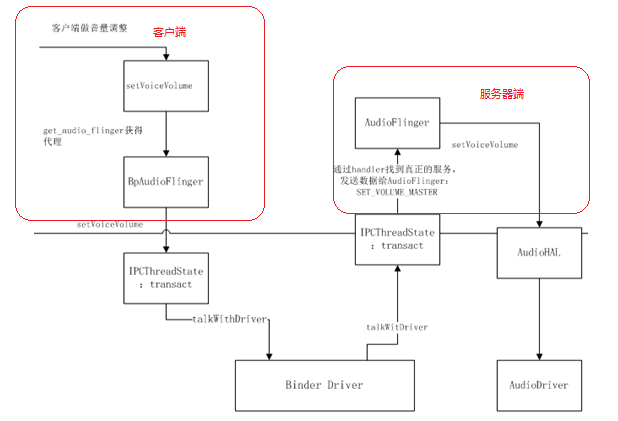
### 4.4.1 Native客户端请求服务

比如，拿音量控制来说，如果客户端请求服务，其流程如下：

|  |
| --- |
| 1，向ServiceManager获取到服务IAudioFlinger实例：get\_audio\_flinger(frameworks/av/libmedia/AudioSystem.cpp) |
| 2，通过上面的步骤分析，现在将服务代理和服务的类框架图结构如下： |

### 4.4.2 Natvie层的C/S通信框架

上面描述了客户端请求AudioFlinger服务代理的过程，那客户端与服务真正通信的过程如何，如下(Native)层结构：



从这里可以看出：

1，客户端没有直接操作服务端，而后通过服务代理来操作，这样子就可以

实现跨进程来调用服务。

2，服务代理实现了逻辑功能，其实质只是与binder通信，通过binder将信

息带个真正的服务器，由服务器来做真正的动作。但因两边都继承了服

务接口，所以函数接口是一致性的。

## 4.5 Java层的Binder框架

### 4.5.1 Java层服务注册

Java层没有提供servicemanager的进程，而是通过ServiceManager类，使用JNI的方式来复用Native层的servicemanager，同时也通过JNI方式封装Native

层的BpBinder,BBinder为Java对象，供Java层使用。

|  |  |
| --- | --- |
| ServiceManager | ->获取native层的servicemanager对象，返回一代理：  ServiceManagerProxy对象(实质通信对象) |
| 操作：  public static IBinder getService(String name)  public static void addService(String name, IBinder service)  实质是调用代理对象的相同方法来实现真正想native层的servicemanager注册服务。 |
| 注册Java层的服务： | |
| PowerManagerService power = new PowerManagerService();对象实际组织形式：    ServiceManager.addService(Context.POWER\_SERVICE,power)  **其中，Power也可以看做是真正服务的Binder。**  //向servicemanager注册的真实的服务。 | |

### 4.5.2 Java层客户端请求服务

|  |
| --- |
| **ContextImpl**中注册POWER\_SERVICE中，其实质是建立了客户端代理与实质服务之间的关联关系： |
| 请求服务：  PowerManager pm = **this**.getSystemService(Context.*POWER\_SERVICE*); |
| 对ContextImpl的详细讲解见：  http://blog.csdn.net/qinjuning/article/details/7310620 |
|  |

### 4.5.3 Java层Binder框架

通过注册和请求服务，其总结归纳如下：

|  |
| --- |
| Java层中的Binder家族如下： |
| FLAG\_ONEWAY为IBinder中定义，表示阻塞还是非阻塞通信。    1，注册服务：将真正服务通过JavaBBinder注册servicemanager进程。  2，请求服务：通过获取其对应的BinderProxy来生成服务的客户单代理Proxy来执行服务。其实质如下： |

### 4.5.1 AIDL

AIDL(Android Interface Definition Language)用于定义C/S体系结构中server端可以提供的服务调用接口，实质Java层的真实服务的代理，其生成的Java类继承了Binder和IxxxxManager，使得其成为一个代理类。对定义aid文件，实质只要定义出服务的业务逻辑就可以了。

例如如下就可以转换为IPowerManager.stub->实际服务代理

|  |
| --- |
|  |

其实际上就可以通过使用aid来实现调用远程服务。

## 4.6 实践应用

|  |
| --- |
| 1，定义自己的service，注册该service到ServiceManager  注:实际移植过程中，也可以不注册给service manager的方式，采用上层直接调用库的方式来控制HW部分，也就是说将service编译成一个 so中，而后被进程应用该so。 |
| 2，了解现有的框架，对native层做优化。 |

## 4.7 总结

按照标准的Android通信框架，其实现通信需要做如下几个步骤：

|  |
| --- |
| 1，服务注册到servicemanager进程中(Binder中对应的handler=0):  （1）找到servicemanager的代理，发送<ADD\_SERVICE，要注册服务对象>  （2）Binder驱动将转交给servicemanager进程处理ADD\_SERVICE，如果添成功，其servicemanager进程中的svclist添加基本信息(最重要的服务名字，对象)，同时写回给binder驱动，binder驱动就为其分配唯一的handler。这样子就注册成功，实质上：handler->标示对应BBinder->真正服务对象(继承BBinder和对应服务接口) |
| 2 向ServiceManager发送请求查询到对应服务  (1)获取ServiceManager的代理  (2)获取服务的Bpbinder对象。请求查询服务请求，带参数名字<name>，此时Binder驱动转发到servicemanager进程查询svclist中是否存在该服务，如果存在的话，就基于该服务binder下对应handler，构建一BpBinder(handler)对象A，写回<A>到servicemanager代理，而后servicemanager代理返回该Bpbinder。  (3)通过装换获取服务代理，实质基于该BpBinder对象构造了BpxxxxService服务代理。  通过(1) (2) (3) ，就形成了一条路通：  Client：BpxxxService<->BpBinder  <->Binder驱动<->  Service:Bbbinder<->xxxService |
| 3， 获取服务代理之后，接着就可以获取需要的请求：  （1）Client实质上是调用执行的逻辑，将需求发送给binder  （2）Binder将数据转发个真正的Service，Service来真正处理 |
| 所以，从整个流程来看：注册，查询服务{由ServiceManager来管理}， 向服务发送请求都是C/S的可以跨进程的通信模型。  同样Java层的也是满足该规则，只是名称叫法不一样罢。也正是因为Native层与Java层都同时存在这一套对应机制，所以就出现了SDK/NDK两种开发模式。 |

# 第5章 Android异步的基础-Handler机制

## 5.1 原理应用分析

Android中的Handler机制实际应用异步更新UI。实质上是生产者消费者模型：

1，Handler 为生产者不断向消息队列投送消息。

2，拥有该消息队列(Looper)的线程不断从消息队列取消息去处理。

所以，要实现异步UI更新，关键点就是Handler要关联主UI线程消息队列即可。

基本模型如下：



|  |
| --- |
| **更详细关系表述：** |
| **Looper**  1，Android系统中主UI线程创建了Looper，所以不需要再创建Looper。  2，在子线程中创建Looper，继承/使用HandlerThread来实现，为子线程开辟了messagequeue。其使用场景：HandlerThread 比较合适处理那些在工作线程执行，需要花费时间偏长的任务。可以通过runOnUIThread来回调到UI。    3，Looper.loop()中处理消息核心函数： |
| **Message**  Message对象创建方法：    Message结构体(赋值)： |
| **Handler**   |  | | --- | | Handler-Looper机制要保证运行成功，最关键的是Handler关联正确的Looper对象。其**关联**的实现是在Handler构造对象时刻做到(常用)：    Android UI异步更新：在主UI线程中创建Handler来实现。 | | **创建Message**    其上面创建了关联Handler（target）基础消息对象，若对于有些信息是自己需要填充。 | | **发送Message**      **发送任务** | | **处理messagequeue中的消息最终函数是调用：**      **处理Message：**  重写handleMessage(msg)函数。  **处理任务：**    在主UI线程创建Handler，那么在推送Runable对象中是可以直接做UI更新操作。子线程中直接回调主线程，还可以使用runOnUiThread。 |   **Handler中引用了MessageQueue，所以在使用Handler时候一定要考虑是否会存在内存泄露问题。** |

## 5.2 Handler使用注意总结

根据Handler机制的原理，使用注意：

|  |
| --- |
| 1，Handler机制功能：可以实现线程间通信，在UI异步更新中，其实现子线程与主UI线程的通信，就可以实现UI的异步更新机制：  1）子线程中执行耗费时间操作。  2）结果通过主UI线程创建的Handler对象来发送消息到主UI线程。  3）主UI线程处理中子线程回调的数据，因handleMessage方法是运行在主线程中的，所以不要执行非常耗时的操作，防止ANR。 |
| 2，当Handler对象在主线程运行时刻，其handlerMessage定义的方法实质也是运行在主UI线程，所以就要避免在该方法中实现耗费时间的任何操作，否则会ANR。  在网络开发中，这也是HttpClient的回调函数中不执行耗时操作原因，要使用AsyncHttpClient，但实际开发中建议不使用。若有执行耗费时间操作，可以考虑使用HandleThrad+onUiThread：在HandlerThread中对应handler中处理耗费时间操作，而后将数据结果通过onUiThread来返回个主线程。 |

# 第6章 APP Framework Layer

## 6.1 应用进程管理器-ActivityManagerService

ActivityManagerService（AMS）为Android系统中最核心的服务之一，可以理解其为进程管理器。其有如下功能：

1，负责四大组件的启动，切换和调度。

2，应用程序的管理和调度器

### 1，AMS的启动（入口）

|  |
| --- |
| AMS(其用户可以使用Activity Manager) Service启动过程：  SystemServer.java中initAndLoop函数，其在SystemServer线程执(在system\_server进程中)行如下操作： |

#### Part1：main函数

main函数执行分为如下几个部分

|  |  |  |  |
| --- | --- | --- | --- |
| 1，创建AThread线程，且启动该线程。该线程名为：ActivityManager，其含有Looper和ActivityManagerService对象，执行函数如下：    2，填充ActivityServiceManager，执行如下方法：    此时类关系图如下：    其中，ContextImpl与ActivityThread为非常重要的两个类：   |  | | --- | | ContextImpl：为Context的实现类，实质是保存资源引用。系统Context实质是存放的与framework-res.apk有关的资源。  几个独立类说明：  1） LoadApk类：保存Apk有关信息。一个Apk文件可以构建一个LoadApk对象  2）Application：理解为一容器，其中含有四大组件，且一个进程可以同时拥有多个Application。一Application对象中含有一个LoadApk对象，即为一对一的关系，且Application构造时候其基于Context。  3）Application实际并不表示进程的概念，且Android中一进程可以包含多个Application对象(ActivityThread中有Application队列看出)。 | | ActivityThread：其实质就是对Handler做了高级封装的修饰类：当在哪个线程中创建ActivityThread对象，那么关联就是该线程的Looper中去。这样可以灵活的绑定到不同的线程中，正因为这种灵活性，使得其承担了管理应用进程的执行调度，组件之间通信和请求等操作。  比如ActivityManagerService实例，其创建mSystemThread是对应SystemServer线程的Looper。 | | ActivityThread. systemMain关键代码：    实质：基于framework-res.apk来创建Application对象，即一个Application环境，让Context指向该环境。所以就system\_server进程中达成：  1)创建ActivityThread主线程  2)一个framework-res.apk对应的Application环境，且系统Context指向该环境。  达到了给system\_server创建一个与应用进程一样的Android运行环境。 |   3. startRunning(null, null, null, null); |

#### Part2: AMS的setSystemProcess函数

其该函数功能是创建一个代表system\_server的ProcessRecord对象，将system\_server加入AMS统一管理。

1，注册服务到servicemanager

activityservicemanager:"activity",meminfo, gfxinfo, dbinfo, cpuinfo, permission

2，执行mSystemThread.installSystemApplicationInfo(info);实质是将系统Context

绑定ApplicationInfo(framework-res.apk对应)对象 ，初始化Android运行环境。

3，将运行环境与一个进程管理结构对应起来，交给AMS统一管理。

|  |
| --- |
|  |
| 相关类分析  ProcessRecord：AMS中进程管理结构  **ApplicationThreead**:其功能是真正执行调度四大组件，Application的服务  IApplicationThread：调度的接口定义  其之间存在关键如下： |

#### Part3：AMS中的systemReady函数

实质是启动Launch功能，其大概可以分为两个主要部分：

|  |
| --- |
| 1，预处理，发送ACTION\_PRE\_BOOT\_COMPLETED广播，一些模块启动对数据库的预处理。当预处理完成之后，设定mSystemReady=true |
| 2，当mSystemRead=true时候，执行SystemServer线程中initAndLoop中ActivityManagerService.self().systemReady实际传入的Runable().run:  1）startSystemUi(contextF); //SystemUI.apk  2）服务启动  3）Wathcdog启动 |
| 3，启动Launch  执行systemReady中的mStackSupervisor.resumeTopActivitiesLocked()-》  mStackSupervisor.mHomeStack.resumeTopActivityLocked()  ->mStackSupervisor.resumeHomeActivity(prev)-> mService.startHomeActivityLocked(mCurrentUser=0)->....  ->startProcessLoacked来创建应用进程：  1，客户端链接Zygote进程的socket，而后通过fork ，Zygote创建新的子进程(VM实例)，其指定类名字为（android.app.ActivityThread）  2，子进程中执行了ActivityThread.main方法创建主线程，即UI线程，而后通过ActivityThread来处理该子进程的线程通信。  当创建了UI子线程之后，就可以用来启动Home了。 |

#### Part4: AMS总结(重点)

|  |
| --- |
| 1，普通应用程序启动过程如下：  1）基于apk包来创建ApplicationInfo对象。  2）基于ApplicationInfo对象来创建ProcessRecord对象（AMS管理的进程结构）。函数：ServiceManager.newProcessRecordLocked  3）基于ProcessRecord对象来创建其Linux环境下真实的Process。其实际流程可以这样子表示：    4） LinuxProcess的子进程执行android.app.ActivityThread.main函数启动第一个子线程（即为主线程[有Looper]），且进入Java环境(Android运行环境)。  5）主线程中又将该Java进程结构ProcessRecord对象绑定了ApplicationThread(实质进程状态管理）对象，从而使得ProcessRecord能实现状态的切换。真正实现了将Linux的Process为其在Android运行环境中创建了对应的process。代码如下：    6） AMS中含有ProcessRecord对象的队列，通过队列来实现进程的管理，调用ProcessRecord对象中ApplicationThread来真正完成对应用进程的调度。 |
| 2，系统服务进程system\_server：  1）其Zygote在Linux环境之下fork创建的子进程，而后子进程中调用SystemServer.main创建了SystemServer(有Looper)线程，在该线程中启动了Native和Java框架层的所有服务。这里得出：所有服务都运行在system\_server进程中。  2）为了AMS能在Android运行环境中也能够对systerm\_server进程做管理，其调用ActivityManagerService.setSystemProcess，为其创建对应的Android环境下的映射进程。  最终，实现了AMS对系统进程/应用进程的统一管理。 |
| 3， 进程主线程  1）普通应用进程，主线程是ActivityThread。其中Handler对象ActivityThread.mH关联ActivityThread的Looper。  2）system\_server进程，主线程是SystemServer。其中Handler对象ActivityThread.mH关联ActivityThread的Looper。  所以，对于ActivityThread类来说，其实现两个功能：  1）对Handler封装的高级修饰类，实际处理四大组件/Application的状态。  2）用来创建主线程 |
| 4， ActivityManagerService，ApplicationThread，ActivityThread的关系：调度/管理过程    其中，最关键是交互界面组件Activity的管理是AMS->ActivityStackSupvirsor->ApplicationThread的函数方法，所以ActivityStack管理了Activity的启动过程。 |
| 5， 进程与Application对象关系  Application对象静态文件实质就是一.apk文件，但一个进程可以含有Application对象，即可以含有多个.apk文件。那怎么配置进程与Application的关系，请查看如下：AndroidManifest.xml。   |  |  | | --- | --- | | 基础：  Android中为每个应用程序创建Linux进程时候，同时为该进程分配了一Linux用户，这样就赋予进程对系统的资源访问权限，保证资源安全性，其对应属性配置就是：android:sharedUserId。  "android.uid.system":可以理解为root，可以访问系统任意资源  "shared"：可以与home/contacts进程共享数据。  Linux用户与进程关系如下：  1）一个进程只有唯一的linux用户。  2）一个linux用户可以同时拥有多个进程。  通过共享userid,拥有相同userid的用户apk可以配置成运行在同一进程当中，因此默认就是可以互相访问任意数据的也可以配置为不同进程当中，彼此之间就像访问自己的数据一样访问彼此的数据库和文件。 | | | 情况1：1进程对应1一个Application，及实际上为一apk创建进程 | default | | 情况2：1进程含有多个Application，即含有多个apk，就多个apk共享进程 | 前提条件：这些apk的android:sharedUserId肯定相同  配置<application android:process>，都指定为同已经 | | Android进程永久驻存 | 设定android:persistent="true"，默认为false的，若为true，则系统启动就一直开启。 | | 应用场景：不同应用进程之间的资源访问方式。  不同应用进程之间共享数据：http://blog.csdn.net/coding\_glacier/article/details/8230159  或者使用ContentProvider | | |

### 2，startActivity 启动分析（后期分析）

非常复杂，后期在分析

### 3，与广播组件的交互

...........

### 4，与service组件的交互

.............

## 6.2 PackageManagerService

本章节主要是了解package manager到底做了哪些事情即可，其具备如下几种功能：

|  |
| --- |
| 1，权限处理，包括对系统和应用定义的Permission和Permission Group的信息增删查 |
| 2，包(APK)处理，包括扫描并安装卸载apk包，查询包的UID，GID，包名及系统默认程序信息。 |
| 3，比较两个包的签名信息 |
| 4，查询Activity，Provider，Receiver，Service四大组件信息 |
| 5，查询Application，Package，Resource，shared Library,Feature的信息 |
| 6，Intent匹配 |

## 6.2 WindowManager System

WindowManager其实实现对窗口的管理。窗体的概念如下：

|  |
| --- |
| ，实质每个窗体中都含有顶层视图就DecorView。 |

而WMS的功能就是：

|  |
| --- |
| 还有添加删除窗体等功能。 |
| 窗体：DecorView |

对于应用程序来说，创建窗口，其实质操作分为如下：

|  |
| --- |
| WindowManager wmManager=(WindowManager) getSystemService(Context.WINDOW\_SERVICE);  WindowManager.LayoutParams lp = new WindowManager.LayoutParams();  wmManager.addView(RootView, lp); |
| lp中的属性定义窗体的类型和显示效果  rootView:其实质定义的窗体显示内容 |
| 移除窗体：wmManager.removeView(View view)  调整View在窗体中的位置，使用函数:  updateViewLayout(View view, ViewGroup.LayoutParams  params) |

### 6.2.1 WindowManager.LayoutParams重要参数分析

LayoutParams参数参考：http://blog.csdn.net/xyz\_fly/article/details/7546210

#### 6.2.1.1 type-窗口类型

Type用来定义窗口的类型，窗口类型分为如下几种：

|  |
| --- |
| **1，应用窗口** |
| Activity: TYPE\_BASE\_APPLICATION  Dialog：TYPE\_APPLICATION  1）构造LayoutParams，默认type=TYPE\_APPLICATION.  2）所有应用窗口都必须对应Activity（基窗口），当Activity创建之后，其对应一基窗口(getWindow()访问)。 |
| **2，子窗口**  子窗口是指该窗口必须要有一个父窗口，父窗口可以是一个应用类型窗口，也可以是任何其他类型的窗口，子窗口的定义,一般子窗口FIRST\_SUB\_WINDOW。 |
| 其特点： B 是 A 的子窗口，当 A 不可见时，B 也会不可见的。如果A不可见时添加B，B 也是不可见的，直到 A 可见为止，B 跟随一起可见。  PopupWindow：TYPE\_APPLICATION\_PANEL |
| **3， 系统窗口**  系统窗口跟应用窗口不同，不需要对应 Activity。跟子窗口不同，不需要有父窗口。一般来讲，系统窗口应该由系统来创建的，例如发生异常，ANR时的提示框，又如系统状态栏，屏保等。但是，Framework 还是定义了一些，可以被应用所创建的系统窗口，如 TYPE\_ TOAST，TYPE \_INPUT \_ METHOD，TYPE \_WALLPAPTER 等等。 |
|  |
| 在使用系统窗口，需要注意权限控制： 以下类型不需要特别声明权限 TYPE \_ TOAST，TYPE \_ DREAM，TYPE \_ INPUT \_ METHOD，TYPE \_ WALLPAPER，TYPE \_ PRIVATE \_ PRESENTATION，TYPE \_ VOICE \_ INTERACTION，TYPE \_ ACCESSIBILITY \_ OVERLAY 以下类型需要声明使用权限：android.permission.SYSTEM \_ ALERT \_ WINDOW TYPE \_ PHONE，TYPE \_ PRIORITY \_ PHONE，TYPE \_ SYSTEM \_ ALERT，TYPE \_ SYSTEM \_ ERROR，TYPE \_ SYSTEM \_ OVERLAY 其他的系统窗口，需要声明权限：android.permission.INTERNAL \_ SYSTEM \_ WINDOW |

#### 6.2.1.2 token-通信基础

token(令牌)一般是指Ibinder对象，实质可以理解为创建BpBinder<->BnBinder对应关系，而后可以适当C/S之间进行通信。其具体如下，其中WindowManager.LayoutParam中定义描述红色：

|  |
| --- |
|  |
| Token的详细说明：  http://bugly.qq.com/bbs/forum.php?mod=viewthread&tid=555&fromuid=6#rd |

### 6.2.2 Activity窗体实例分析

|  |
| --- |
|  |
| 上述就是含有一个Activity基础窗口与系统状态窗口的ViewHierachy图示，其中对于Activity 的View布局如下：    其中DecorView作为根View就是为实际的窗口，这个窗口对象会被添加到WMS，去负责一些列窗体的层次和视图的显示内容。 |
| 其真个在GUI系统显示关系如下表示：    **1，View 系统管理每个窗口中复杂的布局，最终这个 View Hierarchy 最顶端的根 View 会被作为窗口，添加到 Window Manager System 中。**  **2，Window Manager System 管理着所有这些添加的窗口，负责管理这些窗口的层次，显示位置等内容，但是其并不实现绘制功能。**  **3，每个窗口都有一块自己的Surface(一块画布)，应用可以通过Canvas或者OpenGl在上面作画，而后Surface Flinger 负责把这些 Surface 安装Z-order合成输出到FrameBuffer，而后显示出图像给用户。实质是Surface来负责绘制。** |

参考文档：

http://bugly.qq.com/bbs/forum.php?mod=viewthread&tid=555&fromuid=6#rd

## 6.3 Android绘图技术-控件系统

上面章节主要讨论了WMS对窗口的管理和显示，但是其不负责绘制图形。那这部分主要是依附窗口的DecorView下的控件是怎么实现绘制。

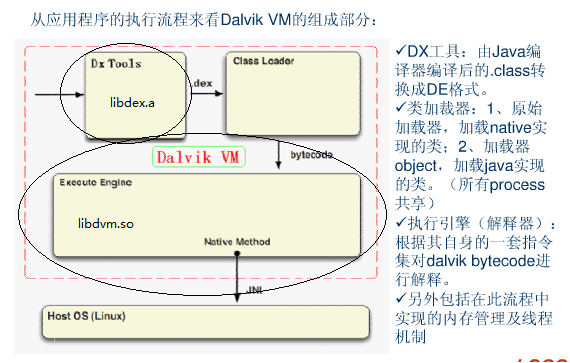
继承于View的视图都可以称为控件。

|  |
| --- |
|  |
| 1，WindowManagerGlobal为WindowManager的实现，所以最后是WindowManagerGlobal与wms做交互，完成窗体的管理。  2， 每一新窗口都对应一个ViewRootImpl对象，实质该对象实现了控件树的根，其实复杂触发控件的测量，布局及绘制过程。 |

# 附录A Android虚拟机

Android 基于JNI机制的Java应用都运行于Dalvik vm，通过Dalvik vm与本地库做交互。所以，了解Android虚拟机对于应用开发有极强的帮助作用。

Dalvik vm虚拟机不同与标准的JVM，其是将.class打包压缩为.dex后在Dalvik虚拟机上做处理。



# 附录B Android JNI框架

logcat整个代码框架

# ~~附录C Android初始化语言(AIL)~~

|  |  |
| --- | --- |
| system\core\init\readme.txt | |
| **Keyword** | **form** |
| Actions | on <trigger> <command>  <command>  <command> |
| Services | service<name><pathname>[ <argument> ]\*  <option>  <option>  ... |
| Options | **critical**  This is a device-critical service. If it exits more than four times in four minutes, the device will reboot into recovery mode.  **disabled**  This service will not automatically start with its class.It must be explicitly started by name.  **setenv <name> <value>**  Set the environment variable <name> to <value> in the launched process.  **socket <name> <type> <perm> [ <user> [ <group> ] ]**  Create a unix domain socket named /dev/socket/<name> and pass  its fd to the launched process. <type> must be "dgram", "stream" or "seqpacket". User and group default to 0.  u**ser <username>**  Change to username before exec'ing this service.  Currently defaults to root. (??? probably should default to nobody) Currently, if your process requires linux capabilities then you cannot use this command. You must instead request the capabilities in-process while still root, and then drop to your desired uid.  **group <groupname> [ <groupname> ]\***  Change to groupname before exec'ing this service. Additional  groupnames beyond the (required) first one are used to set the supplemental groups of the process (via setgroups()).  Currently defaults to root. (??? probably should default to nobody)  **seclabel <securitycontext>**  Change to securitycontext before exec'ing this service.  Primarily for use by services run from the rootfs, e.g. ueventd, adbd. Services on the system partition can instead use policy-defined transitions based on their file security context.  If not specified and no transition is defined in policy, defaults to the init context.  **oneshot**  Do not restart the service when it exits.  **class <name>**  Specify a class name for the service. All services in a  named class may be started or stopped together. A service  is in the class "default" if one is not specified via the  class option.  **onrestart**  Execute a Command (see below) when service restarts. |
| Triggers | **boot**  This is the first trigger that will occur when init starts(after /init.conf is loaded)  **<name>=<value>**  Triggers of this form occur when the property <name> is setto the specific value <value>.  **device-added-<path>**  **device-removed-<path>**  Triggers of these forms occur when a device node is added or removed.  **service-exited-<name>**  Triggers of this form occur when the specified service exits. |
| Commands | **exec <path> [ <argument> ]\***  Fork and execute a program (<path>). This will block until  the program completes execution. It is best to avoid exec  as unlike the builtin commands, it runs the risk of getting  init "stuck". (??? maybe there should be a timeout?)  **export <name> <value>**  Set the environment variable <name> equal to <value> in the  global environment (which will be inherited by all processes  started after this command is executed)  **ifup <interface>**  Bring the network interface <interface> online.  **import <filename>**  Parse an init config file, extending the current configuration.  **hostname <name>**  Set the host name.  **chdir <directory>**  Change working directory.  **chmod <octal-mode> <path>**  Change file access permissions.  **chown <owner> <group> <path>**  Change file owner and group.  **chroot <directory>**  Change process root directory.  **class\_start <serviceclass>**  Start all services of the specified class if they arenot already running.  **class\_stop <serviceclass>**  Stop all services of the specified class if they arecurrently running.  **domainname <name>**  Set the domain name.  **insmod <path>**  Install the module at <path>  **mkdir <path> [mode] [owner] [group]**  Create a directory at <path>, optionally with the given mode, owner, and group. If not provided, the directory is created with permissions 755 and owned by the root user and root group.  **mount <type> <device> <dir> [ <mountoption> ]\***  Attempt to mount the named device at the directory <dir> <device> may be of the form mtd@name to specify a mtd block device by name.<mountoption>s include "ro", "rw", "remount", "noatime", ...  **restorecon <path>**  Restore the file named by <path> to the security context specified in the file\_contexts configuration.Not required for directories created by the init.rc as these are automatically labeled correctly by init.  **setcon <securitycontext>**  Set the current process security context to the specified string.  This is typically only used from early-init to set the init context before any other process is started.  **setenforce 0|1**  Set the SELinux system-wide enforcing status. 0 is permissive (i.e. log but do not deny), 1 is enforcing.  **setkey**  TBD  **setprop <name> <value>**  Set system property <name> to <value>.  **setrlimit <resource> <cur> <max>**  Set the rlimit for a resource.  **setsebool <name> <value>**  Set SELinux boolean <name> to <value>.  <value> may be 1|true|on or 0|false|off  **start <service>**  Start a service running if it is not already running.  **stop <service>**  Stop a service from running if it is currently running.  **symlink <target> <path>**  Create a symbolic link at <path> with the value <target>  **sysclktz <mins\_west\_of\_gmt>**  Set the system clock base (0 if system clock ticks in GMT)  **trigger <event>**  Trigger an event. Used to queue an action from anotheraction.  **wait <path> [ <timeout> ]**  Poll for the existence of the given file and return when found, or the timeout has been reached. If timeout is not specified it currently defaults to five seconds.  **write <path> <string> [ <string> ]\***  Open the file at <path> and write one or more stringsto it with write(2) |
| Section | 每一个service或者action就是一section |

# 附录D 双链表

|  |
| --- |
| struct listnode  {  struct listnode \*next;  struct listnode \*prev;  };  #define node\_to\_item(node, container, member) \  (container \*) (((char\*) (node)) - offsetof(container, member))  #define list\_declare(name) \  struct listnode name = { \  .next = &name, \  .prev = &name, \  }  #define list\_for\_each(node, list) \  for (node = (list)->next; node != (list); node = node->next)  #define list\_for\_each\_reverse(node, list) \  for (node = (list)->prev; node != (list); node = node->prev)  void list\_init(struct listnode \*list);  void list\_add\_tail(struct listnode \*list, struct listnode \*item);  void list\_remove(struct listnode \*item);  #define list\_empty(list) ((list) == (list)->next)  #define list\_head(list) ((list)->next)  #define list\_tail(list) ((list)->prev) |
| void list\_init(struct listnode \*node)  {  node->next = node;  node->prev = node;  }  void list\_add\_tail(struct listnode \*head, struct listnode \*item)  {  item->next = head;  item->prev = head->prev;  head->prev->next = item;  head->prev = item;  }  void list\_remove(struct listnode \*item)  {  item->next->prev = item->prev;  item->prev->next = item->next;  } |

# 附录E Android基础类(Native层)

|  |
| --- |
| **1，Refbase，Sp，Wp** |
| **基本用法：** |
| **研究问题：**一对象可能被多次多地方引用，那么什么时候可以销毁对象，回收内存？引入强引用和弱引用就是为了管理对象的生命周期。 |
| 实际中Android定义了**LightRefBase**来管理对象的生命周期，其实质只是使用到了强引用。 |
| **2，Looper和Handler：**  见第5章  设计思想引入：为了让每个线程具有独立的成员变量，可以采用局部线程存储，即使用ThreadLocal(set ,get方法) |
| **3，ProcessState&&IPCThreadState分析：**   |  | | --- | | **ProcessState分析：**  一进程中保证有唯一的ProcessState对象来表示该进程(sel()获取对象，实现方式就是单例模式)，其中成员变量： | | **IPCThreadState分析：**  实质上为每个线程创建一个LTS的对象，通过该LTS **IPCThreadState对象**来真正实现向Binder设备读写操作，其也是通过self（）对象获取线程的IPCThreadState。其类中，关键成员：  //线程当前所属于的进程对象，主要是拿进程对象中的BinderDevice Fd，  //使用Ioctl的方式与设备进行真正的通信。  **const** sp<ProcessState> mProcess;  //该线程id  **const** pid\_t mMyThreadId;  //线程拥有的读写buffer  Parcel mIn;  Parcel mOut;  与binder通信的框架： |     通过IPCThreadState这个类来真正实现与Binder Driver的通信。 |
| **5，基于Binder通信的IPC基础类(Native层)**   |  | | --- | | 1，通信逻辑：IBinder家族 | | **2，ServiceManager的业务处理逻辑：IServiceManager家族**  在理解该类继承关系之前，先要了解如下两个宏定义：  那么在IServiceManager中：  关于两个宏定义：IInterface.h(framerswork/native/include/binder/\*) |   最终，其IserviceManager的框架如下：    **通过这种形式，就完成通信处理与服务处理逻辑两者的关联：**  其中：  1，BpServiceManager实质上可以理解其服务管理器的一代理，  2，BnServiceManager其是实质对应的服务器，但这里没有派生实现，实际上建立了一个新进程ServiceManager。  通信的模型可以简单为如下：    **注意：**了解了这种C/S通信之后，就很好了解service与它们的客户端的通信模型。 |

# 附录F C++程序设计概要

因C++也是面向对象的一种语言，其扩展了C语言的面向对象部分，但其有些特性与Java存在区别，所以本章节就是研究与Java差异性的部分。

## 1 基础语法部分

### 1.1 C与C++与Java基础部分的差异性

差异性如下：

|  |  |  |  |
| --- | --- | --- | --- |
|  | **c** | **c++** | **Java** |
| **关键字**  **(黑体部分是c语言实现：**  **1，泛装换**  **2，数据和操作封装(struct)**  **3，简写(宏)**  **)** | **void, void\*** | c++中包含了c语言中的所有东西，且c++使用c中的方法和变量等，使用  **extern "C" {**  **......**  **}** | - |
| **typedef** | - |
| **指针** | 引用类型 |
| **struct/union** | class |
| **宏** | - |
| const | final |
| 引用类型/值类型 | - |
| sizeof | - |
| voliate | - |
| extern | import |
| **堆分配** | malloc | new | new |
| free | delete | - |
| 作用域 | - | ：： | - |
| 友元 | - | friend | - |
| **文件组织** | .h  .c(#include"\*.h") | .h  cpp  namespace | package：  .java文件 |
| **预编译** | 见2.1.2小节 | | - |
| **struct：**  **struct其结构体中可以包含：成员变量(含函数指针) 与函数方法。且c++中起还可以使用继承的语法。** | | | |

### 1.2 C与C++中的预编译

预处理过程扫描源代码，对其进行初步的转换，产生新的源代码提供给编译器。可见预处理过程先于编译器对源代码进行处理。

其中，预编译的指令集如下：

|  |  |  |
| --- | --- | --- |
| 指令 | 用途 |  |
| # | 空指令，无任何效果 |  |
| #include | 包含一个源代码文件{控制.h} | #include预处理指令的作用是在指令，处展开被包含的文件(至少支持8层嵌套) |
| #define | 定义宏 |  |
| #undef | 取消已定义的宏 |  |
| #if | 如果给定条件为真，则编译下面代码 |  |
| #ifdef | 如果宏已经定义，则编译下面代码 |  |
| #ifndef | 如果宏没有定义，则编译下面代码 |  |
| #elif | 如果前面的#if给定条件不为真，当前条件为真，则编译下面代码 |  |
| #endif | 结束一个#if……#else条件编译块 |  |
| #error | 停止编译并显示错误信息 |  |

### 1.3 函数

这里主要是讨论函数的形式参数和返回值问题。

**1，形式参数**

在Java中，其实质形式参数都是一种值传递方法，但是c/c++中却有两种方式：值传递和引用传递。那么什么情况之下建议用**引用传递**：

1) 拷贝构造函数

2) 形式参数为一类型对象，使用引用类型。且如果对象只是用于读，可以添加const。

2，函数的返回

在Java中也很简单，实质你可以理解都为返回一个值。但是c/c++中引用的引用，变得较为复杂，如下区别:

|  |
| --- |
| 返回值 |
| int set(){return a;}  此时生成的是变量a的一个拷贝，即生成了一个临时变量，当这个变量使用完毕之后，变量就被销毁了，返回的是a的一份拷贝。所以这种返回不能用作左值运算，如：set() = 5;这是不正确的。 |
| 返回引用：  int& set()      {return a;}  返回的是引用，那么，就相当于返回的是变量a的别名，就可以对该变量操作，如赋值等等。set() = 5;是正确的。  前提保证：变量a不能够被销毁的，一旦销毁别名就没有意义。  **建议使用：当函数返回是一类对象的时候，建议使用返回引用。** |

## 2 面向对象

### 2.1 面向对象特性：封装，继承，多态性

#### 2.1.1封装

##### 2.1.1.1 类的对象

从类的对象创建，初始化，复制与销毁来阐述对象。

**1，对象的创建**

|  |
| --- |
| **与java差异**：构造对象方式C++更加丰富，其  Java中String s = new String("abc");  等价于  C++中 String\* ps = new String("abc");  也就是说Java中引用类型实质就等价于C++的指针。  但C++的创建对象的方式比较复杂，其有两种：  **1，堆上分配对象的空间，即上面指的部分，重点：程序员自己要在适合的时候释放堆空间(delete)，实际也就是要显示的调用析构函数做资源释放。**  **2，栈上分配对象的空间，重点：析构函数的写法要合理。在这种分配方式之下，当方法返回时候，其局部对象会自动调用析构函数释放资源。且构造FI(first in)和析构LO(last out).** |
| Sample[2]：    执行代码：    执行过程说明：  如图：    1，构造过程：  (1)通过默认构造函数在栈中生成t1对象  (2)将t1对象的内容复制给t2  (3)通过默认构造函数创建t3，而后将t2的内容给t3  (4)通过显示构造函数创建t4  且构造过程可以看到：t1，t2，t3的指针区域指向了同一块堆内存，会形成什么问题？ 对象之间的copy方式时候做介绍  2，析构过程：  但函数将要退出时候，c++程序自己执行析构函数，根据构造函数时候入栈顺序t4，t3,t2,t1,那么析构顺序就是：t4,t3,t2,t1。且明显可以看出ptr指向的区域被析构了三次，肯定会出现空指针异常，这就是上面问题的解答。 |

通过上面的例子分析，就会发现:拷贝问题，析构问题，对象成员初始化问题。

首先，引入static关键字修饰的特性：

|  |
| --- |
| **static关键字** |
| **static修饰的成员变量都存放于全局静态区，且该区在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。正因为如此，就必须在编译期间对变量做初始化。根据初始化差异性，**可以分为：  1，全局静态变量/(全局变量):若未初始化，编译器会默认初始化为0  2，类中的静态成员变量：  2.1，静态常量整形变量可以在类定义中赋值，其等价于一常量。  2.2，对其他，因类的初始化时由构造函数执行的，但是对象都没有创建所以没有办法做初始化，所以必须在类的外部初始化，其格式：  **<数据类型><类名>::<静态数据成员名>=<值>**  3，局部静态变量:默认初始化为0，当执行函数中该静态变量之后，才真正初始化。  **因(全局)静态变量/全局变量的初始化都是编译期间就完成，所以其是最先初始化的部分，且其顺序就看其在代码中出现的先后顺序(加载进程就会初始化)。** |

**2，对象的初始化**

1）成员变量按照其在类中声明次序初始化

2）调用类的构造函数

**3，对象的销毁**

对象销毁，实质上都是通过析构函数去实现的，那这种销毁有如下情况：

1）对象作为局部变量，方法退出时候自动调用析构函数

2）堆方式分配对象，通过delete 显示的调用类的析构函数

3）非静态成员对象作为另外一类的成员变量，其所属的类对象销毁时候，该成员对象也销毁

4）全局，静态的只有当程序结束运行了，才销毁对象，即其析构函数只会执行一次。

**4，对象的复制**

上面所描述的是复制方式为“浅复制”，对含有指针类型成员变量，其只是将指针变量所指的地址复制过来，而没有将所指向内容重新复制一份。这种浅复制带来的灾难：对象销毁阶段，指针指向的地址被free了N次，而这会导致地址异常的错误。

所以，对象复制方式：

1）浅复制-类成员变量没有指针，数组等结构。

2）深复制-否则就采用深复制。

至于具体实现方式，在重载章节来介绍。

##### 2.1.1.2 类的访问控制

含有三个关键字：public，protected和private，关系：



##### 2.1.1.3 类的运算符重载

Java中没有定义运算符重载功能的，为c++特有的特性。c++通过重载中实现类似内置基本类型的运算操作。但是相对于提供的运算符重载实现来讲，显得复杂而不直观。

针对内置基本类型，C++已经完整的提供了运算符的实现。但对自定义类型的类，是需要程序员自定义实现的运算符重载。

**1，运算符重载规则**

运算符可以分为一元和二元运算符，其中重载关系如下：

|  |  |  |
| --- | --- | --- |
|  | 运算符 | 重载(是/否) |
| 一元运算符 | ++  -- **[ ]** ->(间接预算符号) | 是 |
| 二元运算符 | +   -  \*  / +=   -=  \*=  /= % %=  <   >  <=  >=   ==  != &&   ||  ^    &     |      ^=    &=    |= >>    <<  **=** | 是，  其中 &,=不需要定义重载，也是可以正常，但=默认是浅拷贝 |
| 运算符 | new     delete  new[ ]  delete[ ] | 是，与操作符new delete的区别 |
| 函数调用操作符 | **()**  **operator()()** | 是，为函数对象使用，实质就局部函数指针功能 |
|  | **：： . .\* ?: sizeof typeid** | 否 |

根据上面描述，归纳如下规则：

|  |  |  |
| --- | --- | --- |
|  | 规则 |  |
| aa@bb | aa.operator@(bb) | 类中定义，函数取一参数 |
| operator(aa,bb) | 类外定义，函数取两个参数 |
| @aa ++ -- -> | aa.operator@() | 类中定义，无参数 |
| operator@(aa) | 类外定义，函数取一参数 |
| aa@ ++ -- [] | aa.operator@(int) | 类中定义，函数取一参数 |
| operator@(aa,int) | 类外定义，函数取两个参数 |

额外介绍new/delete 部分的补充知识：

|  |  |  |
| --- | --- | --- |
| 类别 | 特点 | 操作 |
| 操作符  new  delete | 1，操作符是不可以重载的。  2，调用new是分配空间，且创建一对象，  调用delete是调用析构函数 | new  delete |
| 运算符  new  delete | 1，运算符可以重载  2，new只是分配空间，delete也只是释放空间，其重载函数：  void\* operator new(size\_t sz,argument...)  void operator delete(void\* p) | operator new  operator delete |

2**，运算符重载实现**

本小结以构造Vector为例子来实现重载函数的实现：

|  |
| --- |
|  |
| 分别使用不同的方式去构造对象 |

3**，函数调用()重载**

该重载方式配合算法使用的，其等价于c中向一方法中传递一实现的函数指针。以如下例子来说明：

|  |
| --- |
| **foreach函数的原型定义**：  template<class Iter,class Fct> Fct foreach(Iter b, Iter e,Fct f){  while(b!=e) {f(\*b++);}  return f  } |
| **应用：**有一向量，里面存储了complex类型的数据，现在想通过foreach方式让向量的元素都+ complex(2,3) |
| **思路：**采用函数对象的思想，创建一满足要求的函数对象，如下：  class Add {  private:  complex value;  public：  Add(int real,double i) {val =complex(real,i);}  **operator()(complex & m) {m = m+val}**  }  **说明：**一旦创建Add对象add**，**那么add就可以拥有如下方法：  add.operator()(complex &m)－>add(complex &m)。对于add来说其就有了两个特性：1）add为一对象  2） 对函数调用重载来说，其add可以作为该函数的函数名字(具备等价于C中函数指针的功能)  正因为具备了这两种特性，所以称为函数对象。 |
| **实现：**vector<complex> list; for(list.begin(),list.end(),Add(2,3)); |

#### 2.1.2 继承

整体上来讲，c++中的继承较Java复杂了很多，但继承最基本的特性都保留，只是表示和实现方式不一致。

##### 2.1.2.1 类访问控制关系

在Java中访问控制方式很简单，其提供关键字super来指向基类，但是c++使用(基类：：XX)来访问成员和变量。

其继承方式也包含：public，protected,private。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **public继承方式**   |  |  |  |  | | --- | --- | --- | --- | | 基类成员变量 | public | protected | private | | 子类中该成员变量属性 | public | protected | hide | | 子类访问该成员变量 | yes | yes | no |   **protected继承方式**   |  |  |  |  | | --- | --- | --- | --- | | 基类成员变量 | public | protected | private | | 子类中该成员变量属性 | protected | protected | hide | | 子类访问该成员变量 | yes | yes | no | |
| **private继承方式**   |  |  |  |  | | --- | --- | --- | --- | | 基类成员变量 | public | protected | private | | 子类中该成员变量属性 | private | private | hide | | 子类访问该成员变量 | yes | yes | no | |

##### 2.1.2.2 继承后子类结构

不管是c++与Java中都是通过构造函数去实例化对象，因此构造函数式不可以继承的。所以，要在子类构造函数中去调用基类构造函数，所以 构造序列是：

基类->子类。

**1，单继承**

只继承一个基类方式，实际上Java只能够继承一个直接基类。所以先在单继承基础上去讨论它的特性。

如下例子：

|  |
| --- |
|  |
| 当创建一子类对象之后，大概布局可以如下：    与java相比，**相同点：**  1，子类含有父类的成员变量，只是因访问控制权限的变化，子类不可能直接访问到父类的所有成员。  2，调用子类方法还是作用子类数据，调用父类方法还是作用于父类数据。  **差异点：覆盖实现机制**  Java是如果父类子类有同一个函数，子类重写该函数就是覆盖。但是C++必须：**在虚函数的基础上，子类重新定义该函数的行为就是覆盖，这样子就可以都操作子类的数据变量。** |
| **纯虚函数**：virtual void print() =0; （Java中是抽象函数概念，有abstract关键字修饰）  **抽象类：**含有一个或者多个纯虚函数的类。(Java中是有抽象函数的类) |

**2，多继承**

Java中是通过实现多个接口去实现多继承，但是C++中其可以继承多个类的方式，所以经常就会遇到消除歧义的问题。

因单继承已经讨论了其特性，两种特性都一样，只是其继承了多个类。这里主要讨论**虚基类**：

|  |
| --- |
| 虚基类：类名字前加入virtual |
| 应用：公有的基类在子类中只出现一个，优化了继承结构(类层次) |

#### 2.1.3 多态

实质就是基于覆盖的机制，继承于同一基类的所有子类可以同时去处理同一个消息，如下：

|  |
| --- |
|  |

### 2.2 模板

C++中称为模板，但Java中称为泛型(Jdk1.5引入)。其都具有共同目的：支持通用型设计，使同一份代码能够满足不同类型参数类型的需求。

#### 2.2.1 C++中的泛型(模板)机制

|  |
| --- |
|  |
| **模板：**并非一个实实在在的类或函数，仅仅是一个类和函数的描述。  **C++：**如果不做模板实例化，只要符合模板的语法，其定义的模板都可以编译通过  **Java：**因模板编译时候会使用“边界擦除”技术生成.class的字节码，所以泛型类定义要确保擦除边界后也能够通过 |
| **c++中模板实例化分析(基于例子)：**  1，编译器编译vector<string> v1，其会使用T=string对vector模板实例化，使其成为一具体的实体类vector<string>，且检查该类是否为合法。  2，编译vector<int>v2,也按照1的方法去构建一个实体类vector<int>  **所以，同一模板，使用不同类型参数实例化，对应的为不同的类。** |
| **Java的泛型实例化分析(基于上面的例子)：**  1，编译器编译class vector<T>模板为.class字节码，因做檫除处理，其.class字节码实质都为vector<Object>.  2，vector<Integer>实例化，其是先该引用操作的对象元素类型检查判断是否有错。其实质创建的是一个vector<Object>类的对象。  3，vector<String>也是vector<Object>类的对象。  所以，其泛型是通过"边界擦除"技术实现的，所以Java中实现泛型技术对程序员的一点要求：**消除“边界擦除”技术导致的负面效果，有效方式做“边界限定”** |
|  |

#### 2.2.2 C++中的模板定义

|  |
| --- |
| 函数模板：  template<class T> void out(const T&t) |
| 专门化概念：Java中不存在该概念 |

### 2.3 异常抛出处理

C++中异常处理与Java中的目的和意义基本相同，其处理方式也差不多，但关键字有差异性

|  |  |  |
| --- | --- | --- |
|  | Java | C++ |
| 申明函数可能抛出异常 | throws exception1,exception2 | throw (exception1,exception2) |
| 函数体内抛出 | throw | throw |
| 异常处理 | try {  }catch(exception e){  }**finally** {  } | try {  }catch(exception e){  } |

### 2.4 STL框架

标准库其类似于Java的JDK API包，但是STL的涉及内容远远没有Java强大，与java比较，其含有：

1，集合与Map

2，IO流

3，串

但缺少：网络编程和多线程部分。

在c/c++网络和多线程编程，有几种方式：

|  |  |
| --- | --- |
| 低级形式 | **方式1**，基于操作系统支持的API，用c++封装为线程库和网络库，供调用。  **方法2**，整合第三方库，支持多线程和网络编程。例如Boost线程库，muduo网络库 |
| 高级形式 | 基于**已知应用框架**进行二次开发，如QT。  其中，有两种组织形式：  1）以独立插件形式存在，可以在开发工具中导入该插件，而后配置框架做二次开发，比较常用的就是配合eclipse开发工具。  2）含有该框架的IDE工具，例如Qt Creator,Vs2010等。 |