# TODO

# 为什么要学习Bionic

Bionic库是Android的基础库之一，也是连接Android系统和Linux系统内核的桥梁，Bionic中包含了很多基本的功能模块，这些功能模块基本上都是源于Linux，但是就像青出于蓝而胜于蓝，它和Linux还是有一些不一样的的地方。同时，为了更好的服务Android，Bionic中也增加了一些新的模块，由于本次的主题是Androdi的跨进程通信，所以了解Bionic对我们更好的学习Android的跨进行通信还是很有帮助的。

Android除了使用ARM版本的内核外和传统的x86有所不同，谷歌还自己开发了Bionic库，那么谷歌为什么要这样做那?

# 谷歌为什么使用Bionic库

谷歌使用Bionic库主要因为以下三点:

* 1、谷歌没有使用Linux的GUN Libc，很大一部分原因是因为GNU Libc的授权方式是GPL 授权协议有限制，因为一旦软件中使用了GPL的授权协议，该系统所有代码必须开元。
* 2、谷歌在BSD的C库上的基础上加入了一些Linux特性从而生成了Bionic。Bionic名字的来源就是BSD和Linux的混合。而且不受限制的开源方式，所以在现代的商业公司中比较受欢迎。
* 3、还有就是因为性能的原因，因为Bionic的核心设计思想就是"简单"，所以Bionic中去掉了很多高级功能。这样Bionic库仅为200K左右，是GNU版本体积的一半，这意味着更高的效率和低内存的使用，同时配合经过优化的Java VM Dalvik才可以保证高的性能。

# 三、Bionic库简介

Bionic 音标为 bīˈänik，翻译为"仿生"

Bionic包含了系统中最基本的lib库，包括libc，libm，libdl，libstd++，libthread\_db，以及Android特有的链接器linker。

## 三、Bionic库简介

Bionic 音标为 bīˈänik，翻译为"仿生"

Bionic包含了系统中最基本的lib库，包括libc，libm，libdl，libstd++，libthread\_db，以及Android特有的链接器linker。

## 四、Bionic库的特性

Bionic库的特性很多，受篇幅限制，我挑几个和大家平时接触到的说下

#### (一)、架构

Bionic 当前支持ARM、x86和MIPS执行集，理论上可以支持更多，但是需要做些工作，ARM相关的代码在目录arch-arm中，x86相关代码在arch-x86中，mips相关的代码在arch-mips中。

#### (二)、Linux核心头文件

Bionic自带一套经过清理的Linxu内核头文件，允许用户控件代码使用内核特有的声明(如iotcls，常量等)这些头文件位于目录：

bionic/libc/kernel/common  
bionic/libc/kernel/arch-arm  
bionic/libc/kernel/arch-x86  
bionic/libc/kernel/arch-mips

#### (三)、DNS解析器

虽然Bionic 使用NetBSD-derived解析库，但是它也做了一些修改。

* 1、不实现name-server-switch特性
* 2、读取/system/etc/resolv.conf而不是/etc/resolv.config
* 3、从系统属性中读取服务器地址列表，代码中会查找'net.dns1'，'net.dns2'，等属性。每个属性都应该包含一个DNS服务器的IP地址。这些属性能被Android系统的其它进程修改设置。在实现上，也支持进程单独的DNS服务器列表，使用属性'net.dns1.<pid>'、'net.dns2.<pid>'等，这里<pid> 表示当前进程的ID号。
* 4、在执行查询时，使用一个合适的随机ID(而不是每次+1)，以提升安全性。
* 5、在执行查询时，给本地客户socket绑定一个随机端口号，以提高安全性。
* 6、删除了一些源代码，这些源代码会造成了很多线程安全的问题

#### (四)、二进制兼容性

由于Bionic不与GNU C库、ucLibc，或者任何已知的Linux C相兼容。所以意味着不要期望使用GNU C库头文件编译出来的模块能够正常地动态链接到Bionic

#### (五)、Android特性

Bionict提供了少部分Android特有的功能

##### 1、访问系统特性

Android 提供了一个简单的"共享键/值 对" 空间给系统的中的所有进程，用来存储一定数量的"属性"。每个属性由一个限制长度的字符串"键"和一个限制长度的字符串"值"组成。  
头文件<sys/system\_properties.h>中定义了读系统属性的函数，也定义了键/值对的最大长度。

##### 2、Android用户/组管理

在Android中没有etc/password和etc/groups 文件。Android使用扩展的Linux用户/组管理特性，以确保进程根据权限来对不同的文件系统目录进行访问。  
Android的策略是：

* 1、每个已经安装的的应用程序都有自己的用户ID和组ID。ID从10000（一万）开始，小于10000（一万）的ID留给系统的守护进程。
* 2、tpwnam()能识别一些硬编码的子进程名(如"radio")，能将他们翻译为用户id值，它也能识别"app\_1234"，这样的组合名字，知道将后面的1234和10000(一万)相加，得到的ID值为11234.getgrname()也类似。
* 3、getservent() Android中没有/etc/service，C库在执行文件中嵌入只读的服务列表作为代替，这个列表被需要它的函数所解析。所见文件bionic/libc/netbsd/net/getservent.c和bionic/libc/netbsd/net/service.h。  
  这个内部定义的服务列表，未来可能有变化，这个功能是遗留的，实际很少使用。getservent()返回的是本地数据，getservbyport()和getservbyname()也按照同样的方式实现。
* 4、getprotoent() 在Android中没有/etc/protocel，Bionic目前没有实现getprotocent()和相关函数。如果增加的话，很可能会以getervent()相同的方式。

## 五、Bionic库的模块简介

Bionic目录下一共有5个库和一个linker程序  
5个库分别是:

* 1、libc
* 2、libm
* 3、libdl
* 4、libstd++
* 5、libthread\_db

#### (一)、Libc库

Libc是C语言最基础的库文件，它提供了所有系统的基本功能，这些功能主要是对系统调用的封装，是Libc是应用和Linux内核交流的桥梁，主要功能如下：

* 进程管理：包括进程的创建、调度策略和优先级的调整
* 线程管理：包括线程的创建和销毁，线程的同步/互斥等
* 内存管理：包括内存分配和释放等
* 时间管理：包括获取和保存系统时间、获取当前系统运行时长等
* 时区管理：包括时区的设置和调整等
* 定时器管理：提供系统的定时服务
* 文件系统管理：提供文件系统的挂载和移除功能
* 文件管理：包括文件和目录的创建增删改
* 网络套接字：创建和监听socket，发送和接受
* DNS解析：帮助解析网络地址
* 信号：用于进程间通信
* 环境变量：设置和获取系统的环境变量
* Android Log：提供和Android Log驱动进行交互的功能
* Android 属性：管理一个共享区域来设置和读取Android的属性
* 标准输入/输出：提供格式化的输入/输出
* 字符串：提供字符串的移动、复制和比较等功能
* 宽字符：提供对宽字符的支持。

#### (二)、Libm库

Libm 是数学函数库，提供了常见的数学函数和浮点运算功能，但是Android浮点运算时通过软件实现的，运行速度慢，不建议频繁使用。

#### (三)、libdl库

libdl库原本是用于动态库的装载。很多函数实现都是空壳，应用进程使用的一些函数，实际上是在linker模块中实现。

#### (四)、Libm库

libstd++ 是标准的C++的功能库，但是，Android的实现是非常简单的，只是new，delete等少数几个操作符的实现。

#### (五)、libthread\_db库

libthread\_db 用来支持对多线程的中动态库的调试。

#### (六)、Linker模块

Linux系统上其实有两种并不完全相同的可执行文件

* 一种是静态链接的可执行程序。静态可执行程序包含了运行需要的所有函数，可以不依赖任何外部库来运行。
* 另一种是动态链接的可执行程序。动态链接的可执行程序因为没有包含所需的库文件，因此相对于要小很多。

静态可执行程序用在一些特殊场合，例如，系统初始化时，这时整个系统还没有准备好，动态链接的程序还无法使用。系统的启动程序Init就是一个静态链接的例子。在Android中，会给程序自动加上两个".o"文件，分别是"crtbegin\_static.c"和"certtend\_android.o"，这两个".o"文件对应的源文件位于bionic/libc/arch-common/bionic目录下，文件分别是crtbegin.c和certtend.S。\_start()函数就位于cerbegin.c中。

在动态链接时，execuve()函数会分析可执行文件的文件头来寻找链接器，Linux文件就是ld.so，而Android则是Linker。execuve()函数将会将Linker载入到可执行文件的空间，然后执行Linker的\_start()函数。Linker完成动态库的装载和符号重定位后再去运行真正的可执行文件的代码。

## 六、Bionic库的内存管理函数

#### (一)内存管理函数

对于32位的操作系统，能使用的最大地址空间是4GB，其中地址空间03GB分配给用户进程使用，地址空间3GB4GB由内核使用，但是用户进程并不是在启动时就获取了所有的0~3GB地址空间的访问权利，而是需要事先向内核申请对模块地址空间的读写权利。而且申请的只是地址空间而已，此时并没有分配真是的物理地址。只有当进程访问某个地址时，如果该地址对应的物理页面不存在，则由内核产生缺页中断，在中断中才会分配物理内存并建立页表。如果用户进程不需要某块空间了，可以通过内核释放掉它们，对应的物理内存也释放掉。

但是由于缺页中断会导致运行缓慢，如果频繁的地由内核来分配和释放内存将会降低整个体统的性能，因此，一般操作系统都会在用户进程中提供地址空间的分配和回收机制。用户进程中的内存管理会预先向内核申请一块打的地址空间，称为堆。当用户进程需要分配内存时，由内存管理器从堆中寻找一块空闲的内存分配给用户进程使用。当用户进程释放某块内存时，内存管理器并不会立刻将它们交给内核释放，而是放入空闲列表中，留待下次分配使用。

内存管理器会动态的调整堆的大小，如果堆的空间使用完了，内存管理器会向堆内存申请更多的地址空间，如果堆中空闲太多，内存管理器也会将一部分空间返给内核。

#### (二) Bionic的内存管理器——dlmalloc

dlmalloc是一个十分流行的内存分配器。dlmalloc位于bionic/libc/upstream-dlmalloc下，只有一个C文件malloc.c。由于本次主题是跨进程通信，后续有时间就Android的内存回收单独作为一个课题去讲解，今天就不详细说了，就简单的说下原理。  
dlmalloc的原理:

* dlmalloc内部是以链表的形式将"堆"的空闲空间根据尺寸组织在一起。分配内存时通过这些链表能快速地找到合适大小的空闲内存。如果不能找到满足要求的空闲内存，dlmalloc会使用系统调用来扩大堆空间。
* dlmalloc内存块被称为"trunk"。每块大小要求按地址对齐(默认8个字节)，因此，trunk块的大小必须为8的倍数。
* dlmalloc用3种不同的的链表结构来组织不同大小的空闲内存块。小于256字节的块使用malloc\_chunk结构，按照大小组织在一起。由于尺寸小于的块一共有256/8=32，所以一共使用了32个malloc\_chunk结构的环形链表来组织小于256的块。大小大于256字节的块由结构malloc\_tree\_chunk组成链表管理，这些块根据大小组成二叉树。而更大的尺寸则由系统通过mmap的方式单独分配一块空间，并通过malloc\_segment组成的链表进行管理。
* 当dlmalloc分配内存时，会通过查找这些链表来快速找到一块和要求的尺寸大小最匹配的空闲内存块(这样做事为了尽量避免内存碎片)。如果没有合适大小的块，则将一块大的分成两块，一块分配出去，另一块根据大小再加入对应的空闲链表中。
* 当dlmalloc释放内存时，会将相邻的空闲块合并成一个大块来减少内存碎片。如果空闲块过多，超过了dlmaloc内存的阀值，dlmalloc就开始向系统返回内存。
* dlmalloc除了能管理进程的"堆"空间，还能提供私有堆管理，就是在堆外单独分配一块地址空间，由dlmalloc按照同样的方式进行管理。dlmalloc中用来管理进程的"堆"空间的函数，都带有"dl"前缀，如"dlmalloc"，"dlfree"等，而私有堆的管理函数则带有前缀"msspace\_"，如"msspace\_malloc"

Dalvk虚拟机中使用了dlmalloc进行私有堆管理。

## 七、线程

Bionic中的线程管理函数和通用的Linux版本的实现有很多差异，Android根据自己的需要做了很多裁剪工作。

#### (一)、Bionic线程函数的特性

##### 1、pthread的实现基于Futext，同时尽量使用简单的代码来实现通用操作，特征如下：

* pthread\_mutex\_t，pthread\_cond\_t类型定义只有4字节。
* 支持normal,recursive and error-check 互斥量。考虑到通常大多数的时候都使用normal，对normal分支下代码流程做了很细致的优化
* 目前没有支持读写锁，互斥量的优先级和其他高级特征。在Android还不需要这些特征，但是在未来可能会添加进来。

##### 2、Bionic不支持pthread\_cancel()，因为加入它会使得C库文件明显变大，不太值得，同时有以下几点考虑

* 要正确实现pthread\_cancel()，必须在C库的很多地方插入对终止线程的检测。
* 一个好的实现，必须清理资源，例如释放内存，解锁互斥量，如果终止恰好发生在复杂的函数里面(比如gthosbyname())，这会使许多函数正常执行也变慢。
* pthread\_cancel()不能终止所有线程。比如无穷循环中的线程。
* pthread\_cancel()本身也有缺点，不太容易移植。
* Bionic中实现了pthread\_cleanup\_push()和pthread\_cleanup\_pop()函数，在线程通过调用pthread\_exit()退出或者从它的主函数中返回到时候，它们可以做些清理工作。

##### 3、不要在pthread\_once()的回调函数中调用fork()，这么做会导致下次调用pthread\_once()的时候死锁。而且不能在回调函数中抛出一个C++的异常。

##### 4、不能使用\_thread关键词来定义线程本地存储区。

#### (二)、创建线程和线程的属性

##### 1、创建线程

函数[pthread\_create()](https://link.jianshu.com?t=http:/baike.baidu.com/item/pthread_create)用来创建线程，原型是：

int pthread\_create（（pthread\_t \*thread, pthread\_attr\_t \*attr, void \*（\*start\_routine）（void \*）, void \*arg）

其中，pthread\_t在android中等同于long

typedef long pthread\_t;

* 参数thread是一个指针，pthread\_create函数成功后，会将代表线程的值写入其指向的变量。
* 参数 args 一般情况下为NULL，表示使用缺省属性。
* 参数start\_routine是线程的执行函数
* 参数arg是传入线程执行函数的参数

若线程创建成功，则返回0，若线程创建失败，则返回出错编号。  
PS:要注意的是，pthread\_create调用成功后线程已经创建完成，但是不会立刻发生线程切换。除非调用线程主动放弃执行，否则只能等待线程调度。

##### 2、线程的属性

结构 pthread\_atrr\_t用来设置线程的一些属性，定义如下：

typedef struct

{

uint32\_t flags;

void \* stack\_base; //指定栈的起始地址

size\_t stack\_size; //指定栈的大小

size\_t guard\_size;

int32\_t sched\_policy; //线程的调度方式

int32\_t sched\_priority; //线程的优先级

}

使用属性时要先初始化，函数原型是：

int pthread\_attr\_init(pthread\_attr\_t\* attr)

通过pthread\_attr\_init()函数设置的缺省属性值如下：

int pthread\_attr\_init(pthread\_attr\_t\* attr){

attr->flag=0;

attr->stack\_base=null;

attr->stack\_szie=DEFAULT\_THREAD\_STACK\_SIZE; //缺省栈的尺寸是1MB

attr0->quard\_size=PAGE\_SIZE; //大小是4096

attr0->sched\_policy=SCHED\_NORMAL; //普通调度方式

attr0->sched\_priority=0; //中等优先级

return 0；

}

下面介绍每项属性的含义。

* 1、flag 用来表示线程的分离状态  
  Linux线程有两种状态:分离(detch)状态和非分离(joinable)状态，如果线程是非分离状态(joinable)状态，当线程函数退出时或者调用pthread\_exit()时都不会释放线程所占用的系统资源。只有当调用了pthread\_join()之后这些资源才会释放。如果是分离(detach)状态的线程，这些资源在线程函数退出时调用pthread\_exit()时会自动释放
* 2、stack\_base: 线程栈的基地址
* 3、stack\_size: 线程栈的大小。基地址和栈的大小。
* 4、guard\_size: 线程的栈溢出保护区大小。
* 5、sched\_policy：线程的调度方式。  
  线程一共有3中调度方式：SCHED\_NORMAL，SCHED\_FIFO，SCHED\_RR。其中SCHED\_NORMAL代表分时调度策略，SCHED\_FIFO代表实时调度策略，先到先服务，一旦占用CPU则一直运行，一直运行到有更高优先级的任务到达，或者自己放弃。SCHED\_RR代表实时调度策略:时间片轮转，当前进程时间片用完，系统将重新分配时间片，并置于就绪队尾。
* 6、sched\_priority:线程的优先级。

Bionic虽然也实现了pthread\_attr\_setscope()函数，但是只支持PTHREAD\_SCOP\_SYSTEM属性，也就意味着Android线程将在全系统的范围内竞争CPU资源。

##### 3、退出线程的方法

###### (1)、调用pthread\_exit函数退出

一般情况下，线程运行函数结束时线程才退出。但是如果需要，也可以在线程运行函数中调用pthread\_exit()函数来主动退出线程运行。函数原型如下：

void pthread\_exit( void \* retval) ;

其中参数retval用来设置返回值

###### (2)、设备布尔的全局变量

但是如果希望在其它线程中结束某个线程？前面介绍了Android不支持pthread\_cancel()函数，因此，不能在Android中使用这个函数来结束线程。通俗的方法是，如果线程在一个循环中不停的运行，可以在每次循环中检查一个初始值为false的全局变量，一旦这个变量的值为ture，则主动退出，这样其它线程就可以铜鼓改变这个全局变量的值来控制线程的退出，示例如下：

bool g\_force\_exit =false;

void \* thread\_func(void \*){

for(;;){

if(g\_force\_exit){

break;

}

.....

}

return NULL;

}

int main(){

.....

q\_force\_exit=true； //青坡线程退出

}

这种方法实现起来简单可靠，在编程中经常使用。但它的缺点是：如果线程处于挂起等待状态，这种方法就不适用了。  
另外一种方式是使用pthread\_kill()函数。pthread\_kill()函数的作用不是"杀死"一个线程，而是给线程发送信号。函数如下：

int pthread\_kill(pthread tid,int sig);

即使线程处于挂起状态，也可以使用pthead\_kill()函数来给线程发送消息并使得线程执行处理函数，使用pthread\_kill()函数的问题是：线程如果在信号处理函数中退出，不方便释放在线程的运行函数中分配的资源。

###### (3)、通过管道

更复杂的方法是：创建一个管道，在线程运行函数中对管道"读端"用select()或epoll()进行监听，没有数据则挂起线程，通过管道的"写端"写入数据，就能唤起线程，从而释放资源，主动退出。

##### 4、线程的本地存储TLS

线程本地存储(TLS)用来保存、传递和线程有关的数据。例如在前面说道的使用pthread\_kill()函数关闭线程的例子中，需要释放的资源可以使用TLS传递给信号处理函数。

###### (1)、TLS介绍

TLS在线程实例中是全局可见的，对某个线程实例而言TLS是这个线程实例的私有全局变量。同一个线程运行函数的不同运行实例，他们的TLS是不同的。在这个点上TLS和线程的关系有点类似栈变量和函数的关系。栈变量在函数退出时会消失，TLS也会在线程结束时释放。Android实现了TLS的方式是在线程栈的顶开辟了一块区域来存放TLS项，当然这块区域不再受线程栈的控制。

TLS内存区域按数组方式管理，每个数组元素称为一个slot。Android 4.4中的TLS一共有128 slot，这和Posix中的要求一致(Android 4.2是64个)

###### (2)、TLS注意事项

* TLS变量的数量有限，使用前要申请一个key，这个key和内部的slot关联一起，使用完需要释放。  
  申请一个key的函数原型：

int pthread\_key\_create(pthread\_key\_t \*key，void (\*destructor\_function) (void \*) );

pthread\_key\_create()函数成功返回0，参数key中是分配的slot，如果将来放入slot中的对象需要在线程结束的时候由系统释放，则需要提供一个释放函数，通过第二个函数destructor\_function传入。

* 释放 TLS key的函数原型是：

int pthread\_key\_delete ( pthread\_key\_t) ;

pthread\_key\_delete()函数并不检查当前是否还有线程正在使用这个slot，也不会调用清理函数，只是将slot释放以供下次调用pthread\_key\_create()使用。

* 利用TLS保存数据中函数原型：

int pthread\_setspecific(pthread\_key\_t key，const void \*value) ;

* 读取TLS保存数据中的函数原型：

void \* pthread\_getsepcific (pthread\_key\_t key);

##### 5、线程的互斥量(Mutex)函数

Linux线程提供了一组函数用于线程间的互斥访问，Android中的Mutex类实质上是对Linux互斥函数的封装，互斥量可以理解为一把锁，在进入某个保护区域前要先检查是否已经上锁了。如果没有上锁就可以进入，否则就必须等待，进入后现将锁锁上，这样别的线程就无法再进入了，退出保护区后腰解锁，其它线程才可以继续使用

###### (1)、Mutex在使用前需要初始化

初始化函数是:

int pthread\_mutex\_init(pthread\_mutext\_t \*mutex, const pthread\_mutexattr\_t \*attr);

成功后函数返回0，metex被初始化成未锁定的状态。如果参数attr为NULL，则使用缺省的属性MUTEX\_TYPE-BITS\_NORMAL。  
互斥量的属性主要有两种，类型type和范围scope，设置和获取属性的函数如下：

int pthread\_mutexattr\_settype (pthread\_mutexattr\_t \* attr, type);

int pthread\_mutexattr\_gettype (const pthread\_mutexattr\_t \* attr, int \*type);

int pthread\_mutexattr\_getpshared(pthread\_mutexattr\_t \*attr, int \*pshared );

int pthread\_mutexattrattr\_ setpshared (pthread\_mutexattr\_t \*attr,int pshared);

互斥量Mutex的类型(type) 有3种

* PTHREAD\_MUTEX\_NORMAL:该类型的的互斥量不会检测死锁。如果线程没有解锁(unlock)互斥量的情况下再次锁定该互斥量，会产生死锁。如果线程尝试解锁由其他线程锁定的互斥量会产生不确定的行为。如果尝试解锁未锁定的互斥量，也会产生不确定的行为。\*\* 这是Android目前唯一支持的类型 \*\*。
* PTHREAD\_MUTEX\_ERRORCHECK:此类型的互斥量可提供错误检查。如果线程在没有解锁互斥量的情况下尝试重新锁定该互斥量，或者线程尝试解锁的互斥量由其他线程锁定。\*\* Android目前不支持这种类型 \*\* 。
* PTHREAD\_MUTEX\_RECURSIVE。如果线程没有解锁互斥量的情况下重新锁定该互斥量，可成功锁定该互斥量，不会产生死锁情况，但是多次锁定该互斥量需要进行相同次数的解锁才能释放锁，然后其他线程才能获取该互斥量。如果线程尝试解锁的互斥量已经由其他线程锁定，则会返回错误。如果线程尝试解锁还未锁定的互斥量，也会返回错误。\*\* Android目前不支持这种类型 \*\* 。

互斥量Mutex的作用范围(scope) 有2种

* PTHREAD\_PROCESS\_PRIVATE:互斥量的作用范围是进程内，这是缺省属性。
* PTHREAD\_PROCESS\_SHARED:互斥量可以用于进程间线程的同步。Android文档中说不支持这种属性，但是实际上支持，在audiofliger和surfacefliger都有用到，只不过在持有锁的进程意外死亡的情况下，互斥量(Mutex)不能释放掉，这是目前实现的一个缺陷。

##### 6、线程的条件量(Condition)函数

###### (1)为什么需要条件量Condition函数

条件量Condition是为了解决一些更复杂的同步问题而设计的。考虑这样的一种情况，A和B线程不但需要互斥访问某个区域，而且线程A还必须等待线程B的运行结果。如果仅使用互斥量进行保护，在线程B先运行的的情况下没有问题。但是如果线程A先运行，拿到互斥量的锁，往下忘无法进行。

条件量就是解决这类问题的。在使用条件量的情况下，如果线程A先运行，得到锁以后，可以使用条件量的等待函数解锁并等待，这样线程B得到了运行的机会。线程B运行完以后通过条件量的信号函数唤醒等待的线程A，这样线程A的条件也满足了，程序就能继续执行力额。

###### (2)Condition函数

1️⃣ 条件量在使用前需要先初始化，函数原型是：

int pthread\_cond\_init(pthread\_cond\_t \*cond, const pthread\_condattr \*attr);

使用完需要销毁，函数原型是：

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

条件量的属性只有 "共享(share)" 一种，下面是属性相关函数原型，下面是属性相关的函数原型：

int pthread\_condattr\_init(pthread\_condattr\_t \*attr);

int pthread\_condattr\_getpshared(pthread\_condattr\_t \*attr,int \*pshared);

int pthread\_condattr\_setpshared(pthread\_condattr\_t \*attr,int pshared)

int pthread\_condattr\_destroy (pthread\_condattr\_t \*\_\_attr);

"共享(shared)" 属性的值有两种

* PTHREAD\_PROCESS\_PRIVATE：条件量的作用范围是进程内，这是缺省的属性。
* PTHREAD\_PROCESS\_SHARED：条件量可以用于进程间线程同步。

2️⃣条件量的等待函数的原型如下：

int pthread\_cond\_wait (pthread\_cond\_t \*\_\_restrict \_\_cond,pthread\_mutex\_t \*\_\_restrict \_\_mutex);

int pthread\_cond\_timedwait (pthread\_cond\_t \*\_\_restrict \_\_cond,pthread\_mutex\_t \*\_\_restrict \_\_mutex, \_\_const struct timespec \*\_\_restrict \_\_abstime);

条件量的等待函数会先解锁互斥量，因此，使用前一定要确保mutex已经上锁。锁上后线程将挂起。pthread\_cond\_timedwait()用在希望线程等待一段时间的情况下，如果时间到了线程就会恢复运行。

3️⃣ 可以使用函数pthread\_cond\_signal()来唤醒等待队列中的一个线程，原型如下：

int pthread\_cond\_signal (pthread\_cond\_t \*\_\_cond);

也可以通过pthread\_cond\_broadcast()唤醒所有等待的线程

int pthread\_cond\_broadcast (pthread\_cond\_t \*\_\_cond);

#### (三)、Futex同步机制

* Futex 是 fast userspace mutext的缩写，意思是快速用户控件互斥体。这里讨论Futex是因为在Android中不但线程函数使用了Futex，甚至一些模块中也直接使用了Futex作为进程间同步手段，了解Futex的原理有助于我们理解这些模块的运行机制。
* Linux从2.5.7开始支持Futex。在类Unix系统开发中，传统的进程同步机制都是通过对内核对象进行操作来完成，这个内核对象在需要同步的进程中都是可见的。这种同步方法因为涉及用户态和内核态的切换，效率比较低。使用了传统的同步机制时，进入临界区即使没有其他进程竞争也会切到内核态检查内核同步对象的状态，这种不必要的切换明显降低了程序的执行效率。
* Futex就是为了解决这个问题而设计的。Futex是一种用户态和内核态混合的同步机制，使用Futex同步机制，如果用于进程间同步，需要先调用mmap()创建一块共享内存，Futex变量就位于共享区。同时对Futex变量的操作必须是原子的，当进程驶入进入临界区或者退出临界区的时候，首先检查共享内存中的Futex变量，如果没有其他进程也申请了使用临界区，则只修改Futex变量而不再执行系统调用。如果同时有其他进程也申请使用临界区，还是需要通过系统调用去执行等待或唤醒操作。这样通过用户态的Futex变量的控制，减少了进程在用户态和内核态之间切换的次数，从而最大程度的降低了系统同步的开销。

##### 1、Futex的系统调用

在Linux中，Futex系统调用的定义如下：

#define \_NR\_futex 240

(1) Fetex系统调用的原型是：

int futex(int \*uaddr, int cp, int val, const struct timespec \*timeout, int \*uaddr2, int val3);

* uaddr是Futex变量，一个共享的整数计数器。
* op表示操作类型，有5中预定义的值，但是在Bionic中只使用了下面两种：① FUTEX\_WAIT,内核将检查uaddr中家属器的值是否等于val，如果等于则挂起进程，直到uaddr到达了FUTEX\_WAKE调用或者超时时间到。②FUTEXT\_WAKE：内核唤醒val个等待在uaddr上的进程。
* val存放与操作op相关的值
* timeout用于操作FUTEX\_WAIT中，表示等待超时时间。
* uaddr2和val3很少使用。

###### (1) 在Bionic中，提供了两个函数来包装Futex系统调用：

extern int \_futex\_wait(volatile void \*ftx,int val, const struct timespec \*timespec );

extern int \_futex\_wake(volatile void \*ftx, int count);

###### (2) Bionic还有两个类似的函数，它们的原型如下：

extern int \_futex\_wake\_ex(volatile void \*ftx,int pshared,int val);

extern int \_futex\_wait\_ex(volatile void \*fex,int pshared,int val, const stuct timespec \*timeout);

这两个函数多了一个参数pshared,pshared的值为true 表示wake和wait操作是用于进程间的挂起和唤醒；值为false表示操作于进程内线程的挂起和唤醒。当pshare的值为false时，执行Futex系统调用的操作码为

FUTEX\_WAIT|FUTEX\_PRIVATE\_FLAG

内核如何检测到操作有FUTEX\_PRIVATE\_FLAG标记，能以更快的速度执行七挂起和唤醒操作。  
\_futex\_wait 和\_futex\_wake函数相当于pshared等于true的情况。

###### (3) 在Bionic中，提供了两个函数来包装Futex系统调用：

extern int \_futex\_syscall3(volatile void \*ftx,int pshared,int val);

extern int \_futex\_syscall4(volatile void \*ftx,int pshared,int val, const struct timespec \*timeout);

\_futex\_syscall3()相当于 \_futex\_wake()，而 \_futex\_system4()相当于 \_futex\_wait()。这两个函数与前面的区别是能指定操作码op作为参数。操作码可以是FUTEX\_WAIT\_FUTEX\_WAKE或者它们和FUTEX\_PRIVATE\_FLAG的组合。

##### 2、Futex的用户态操作

Futex的系统调用FUTEX\_WAIT和FUTEX\_WAKE只是用来挂起或者唤醒进程，Futex的同步机制还包括用户态下的判断操作。用户态下的操作没有固定的函数调用，只是一种检测共享变量的方法。Futex用于临界区的算法如下：

* 首先创建一个全局的整数变量作为Futex变量，如果用于进程间的同步，这个变量必须位于共享内存。Futex变量的初始值为0。
* 当进程或线程尝试持有锁的时候，检查Futex变量的值是否为0，如果为0，则将Futex变量的值设为1，然后继续执行；如果不为0，将Futex的值设为2以后，执行FUTEX\_WAIT 系统调用进入挂起等待状态。
* Futex变量值为0表示无锁状态，1表示有锁无竞争的状态，2表示有竞争的状态。
* 当进程或线程释放锁的时候，如果Futex变量的值为1，说明没有其他线程在等待锁，这样讲Futex变量的值设为0就结束了；如果Futex变量的值2，说明还有线程等待锁，将Futex变量值设为0，同时执行FUTEX\_WAKE()系统调用来唤醒等待的进程。

对Futex变量操作时，比较和赋值操作必须是原子的。

# Bionic

（Android的Linux内核库）

Android除了使用的是ARM版本的内核外和传统的x86有所不同外，重要的是Google为Linux内核增强了不少东西，自己开发了 Bionic库，同时又贡献给Linux社区了。首先GNU的内核在体积和运行效率上不适合移动设备，系统核心组件都是以动态库的形式驻留在每个进程中， 运行效率和内存占用都是十分重要的问题。Google开发了一个自定义的库名为**Bionic**，以BSD许可形式开源

Bionic库仅为200KB大小是GNU版本体积的一半，这意味着更高的效率和低内存占用，同时配合经过优化的Java VM Dalvik才可以保证高的性能。Bionic不支持一些特性比如宽字节对unicode

bionic.libc

# REF

Android跨进程通信IPC之2——Bionic

**https://www.jianshu.com/p/25a908c7eefa**