阿里巴巴 Android 面试题集（答案解析）

目录

第一章计算机基础面试题

第一节、网络面试题

一、HTTP/HTTPS （⭐⭐⭐）

1、HTTP 与 HTTPS 有什么区别？

HTTPS 是一种通过计算机网络进行安全通信的传输协议。HTTPS 经由 HTTP 进行通信，但利用 SSL/TLS 来加密数据包。HTTPS 开发的主要目的，是提供对网站服务器的身份 认证，保护交换数据的隐私与完整性。

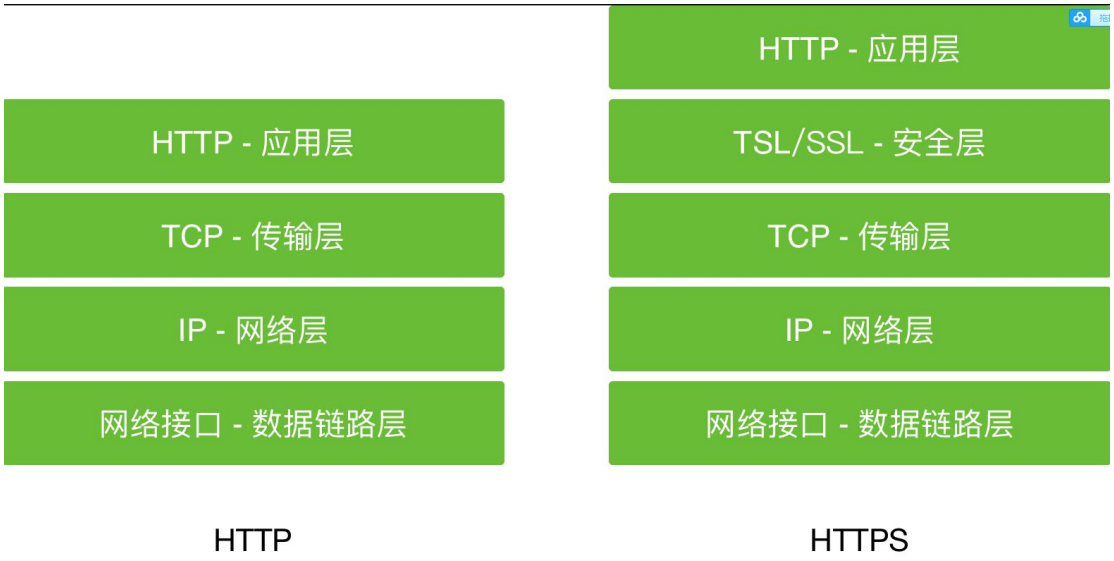
HTPPS 和 HTTP 的概念：

HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。 它是一个 URI scheme（抽象标识符体系），句法类同 http: 体系。用于安全的 HTTP 数据传输。https:URL 表明它使用了 HTTP，但 HTTPS 存在不同于 HTTP 的默认端口及一个加密/身份验证层（在 HTTP 与 TCP 之间）。这个系统的最初研发由网景公司进行，提供了身份验证与加密通讯方法，现在它被广泛用于万维网上安全敏感的通讯，例如交易支付方面。

超文本传输协议 (HTTP-Hypertext transfer protocol) 是一种详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

https 协议需要到 ca 申请证书，一般免费证书很少，需要交费。http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议 http 和 https 使用的是完全不同的连接方式用的端口也不一样,前者是 80,后者是 443。http 的连接很简单,是无状态的 HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议 要比 http 协议安全 HTTPS 解决的问题：1．信任主机的问题. 采用 https 的 server 必须从 CA 申请一个用于证明服务器用途类型的证书. 改证书只有用于对应的 server 的时候,客户度才信任次主机 2．防止通讯过程中的数据的泄密和被窜改

如下图所示，可以很明显的看出两个的区别：



注：TLS 是 SSL 的升级替代版，具体发展历史可以参考传输层安全性协议。

HTTP 与 HTTPS 在写法上的区别也是前缀的不同，客户端处理的方式也不同，具体说来：

如果 URL 的协议是 HTTP，则客户端会打开一条到服务端端口 80（默认）的连接，并向其发送老的 HTTP 请求。 如果 URL 的协议是 HTTPS，则客户端会打开一条到服务端端口 443（默认）的连接，然后与服务器握手，以二进制格式与服务器交换一些 SSL 的安全参数，附上加密的 HTTP 请求。 所以你可以看到，HTTPS 比 HTTP 多了一层与 SSL 的连接，这也就是客户端与服务端 SSL 握手的过程，整个过程主要完成以下工作：

交换协议版本号 选择一个两端都了解的密码 对两端的身份进行认证 生成临时的会话密钥，以便加密信道。 SSL 握手是一个相对比较复杂的过程，更多关于 SSL 握手的过程细节可以参考 TLS/SSL 握手过程

SSL/TSL 的常见开源实现是 OpenSSL，OpenSSL 是一个开放源代码的软件库包，应用程序可以使用这个包来进行安全通信，避免窃听，同时确认另一端连接者的身份。这个包广泛被应用在互联网的网页服务器上。 更多源于 OpenSSL 的技术细节可以参考 OpenSSL。

2、Http1.1 和 Http1.0 及 2.0 的区别？

HTTP1.0 和 HTTP1.1 的一些区别

HTTP1.0 最早在网页中使用是在 1996 年，那个时候只是使用一些较为简单的网页上和网络请求上，而 HTTP1.1 则在 1999 年才开始广泛应用于现在的各大浏览器网络请求中，同时 HTTP1.1 也是当前使用最为广泛的 HTTP 协议。 主要区别主要体现在：

1、缓存处理，在 HTTP1.0 中主要使用 header 里的 If-Modified-Since,Expires 来做为缓存判断的标准，HTTP1.1 则引入了更多的缓存控制策略例如 Entity tag，If-Unmodified-Since, If-Match, If-None-Match 等更多可供选择的缓存头来控制缓存策略。

2、带宽优化及网络连接的使用，HTTP1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

3、错误通知的管理，在 HTTP1.1 中新增了 24 个错误状态响应码，如 409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。

4、Host 头处理，在 HTTP1.0 中认为每台服务器都绑定一个唯一的 IP 地址，因此，请求消息中的 URL 并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个 IP 地址。HTTP1.1 的请求消息和响应消息都应支持 Host 头域，且请求消息中如果没有 Host 头域会报告一个错误（400 Bad Request）。

5、长连接，HTTP 1.1 支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，减少了建立和关闭连接的消耗和延迟，在 HTTP1.1 中默认开启 Connection： keep-alive，一定程度上弥补了 HTTP1.0 每次请求都要创建连接的缺点。

SPDY

在讲 Http1.1 和 Http2.0 的区别之前，还需要说下 SPDY，它是 HTTP1.x 的优化方案：

2012 年 google 如一声惊雷提出了 SPDY 的方案，优化了 HTTP1.X 的请求延迟，解决了 HTTP1.X 的安全性，具体如下：

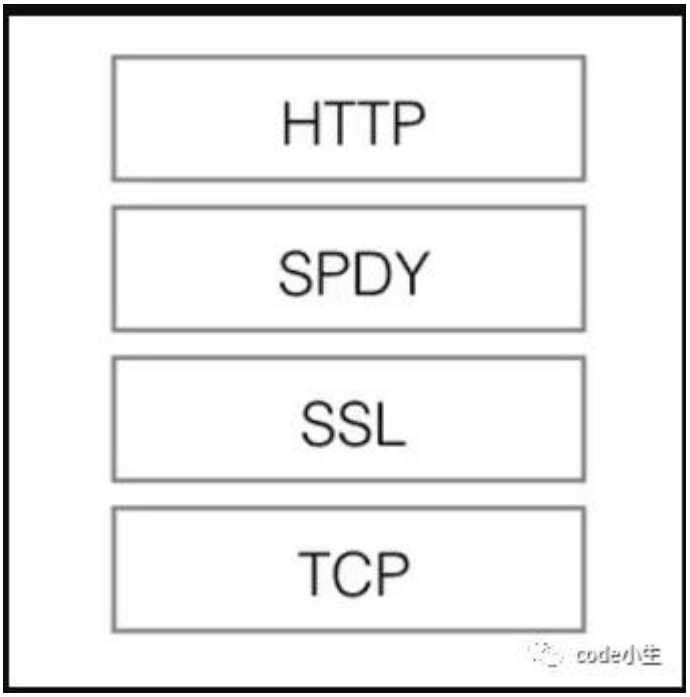
1、降低延迟，针对 HTTP 高延迟的问题，SPDY 优雅的采取了多路复用（multiplexing）。多路复用通过多个请求 stream 共享一个 tcp 连接的方式，解决了 HOL blocking 的问题，降低了延迟同时提高了带宽的利用率。

2、请求优先级（request prioritization）。多路复用带来一个新的问题是，在连接共享的基础之上有可能会导致关键请求被阻塞。SPDY 允许给每个 request 设置优先级，这样重要的请求就会优先得到响应。比如浏览器加载首页，首页的 html 内容应该优先展示，之后才是各种静态资源文件，脚本文件等加载，这样可以保证用户能第一时间看到网页内容。

3、header 压缩。前面提到 HTTP1.x 的 header 很多时候都是重复多余的。选择合适的压缩算法可以减小包的大小和数量。

4、基于 HTTPS 的加密协议传输，大大提高了传输数据的可靠性。

5、服务端推送（server push），采用了 SPDY 的网页，例如我的网页有一个 sytle.css 的请求，在客户端收到 sytle.css 数据的同时，服务端会将 sytle.js 的文件推送给客户端，当客户端再次尝试获取 sytle.js 时就可以直接从缓存中获取到，不用再发请求了。SPDY 构成图：



SPDY 位于 HTTP 之下，TCP 和 SSL 之上，这样可以轻松兼容老版本的 HTTP 协议(将 HTTP1.x 的内容封装成一种新的 frame 格式)，同时可以使用已有的 SSL 功能。

HTTP2.0 和 HTTP1.X 相比的新特性

新的二进制格式（Binary Format），HTTP1.x 的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认 0 和 1 的组合。基于这种考虑 HTTP2.0 的协议解析决定采用二进制格式，实现方便且健壮。

多路复用（MultiPlexing），即连接共享，即每一个 request 都是是用作连接共享机制的。一个 request 对应一个 id，这样一个连接上可以有多个 request，每个连接的 request 可以随机的混杂在一起，接收方可以根据 request 的 id 将 request 再归属到各自不同的服务端请求里面。

header 压缩，如上文中所言，对前面提到过 HTTP1.x 的 header 带有大量信息，而且每次都要重复发送，HTTP2.0 使用 encoder 来减少需要传输的 header 大小，通讯双方各自 cache 一份 header fields 表，既避免了重复 header 的传输，又减小了需要传输的大小。

服务端推送（server push），同 SPDY 一样，HTTP2.0 也具有 server push 功能。

3、Https 请求慢的解决办法

1、不通过 DNS 解析，直接访问 IP

2、解决连接无法复用

http/1.0 协议头里可以设置 Connection:Keep-Alive 或者 Connection:Close，选择是否允许在一定时间内复用连接（时间可由服务器控制）。但是这对 App 端的请求成效不大，因为 App 端的请求比较分散且时间跨度相对较大。

方案 1.基于 tcp 的长连接（主要） 移动端建立一条自己的长链接通道，通道的实现是基于 tcp 协议。基于 tcp 的 socket 编程技术难度相对复杂很多，而且需要自己定制协议。但信息的上报和推送变得更及时，请求量爆发的时间点还能减轻服务器压力（避免频繁创建和销毁连接）

方案 2.http long-polling 客户端在初始状态发送一个 polling 请求到服务器，服务器并不会马上返回业务数据，而是等待有新的业务数据产生的时候再返回，所以链接会一直被保持。一但结束当前连接，马上又会发送一个新的 polling 请求，如此反复，保证一个连接被保持。存在问题： 1）增加了服务器的压力 2）网络环境复杂场景下，需要考虑怎么重建健康的连接通道 3)polling 的方式稳定性不好 4)polling 的 response 可能被中间代理 cache 住 ……

方案 3.http streaming 和 long-polling 不同的是，streaming 方式通过再 server response 的头部增加“Transfer Encoding:chuncked”来告诉客户端后续还有新的数据到来 存在问题：1）有些代理服务器会等待服务器的 response 结束之后才将结果推送给请求客户端。streaming 不会结束 response 2）业务数据无法按照请求分割 ……

方案 4.web socket 和传统的 tcp socket 相似，基于 tcp 协议，提供双向的数据通道。它的优势是提供了 message 的概念，比基于字节流的 tcp socket 使用更简单。技术较新，不是所有浏览器都提供了支持。

3、解决 head of line blocking

它的原因是队列的第一个数据包（队头）受阻而导致整列数据包受阻

使用 **http pipelining**，确保几乎在同一时间把 request 发向了服务器

4、Http 的 request 和 response 的协议组成

1、Request 组成

客户端发送一个 HTTP 请求到服务器的请求消息包括以下格式：

请求行（request line）、请求头部（header）、空行和请求数据四个部分组成。

请求行以一个方法符号开头，以空格分开，后面跟着请求的 URI 和协议的版本。

Get 请求例子

```
GET /562f25980001b1b106000338.jpg HTTP/1.1

Host    img.mukewang.com

User-Agent  Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36

Accept  image/webp,image/*,*/*;q=0.8

Referer http://www.imooc.com/

Accept-Encoding gzip, deflate, sdch

Accept-Language zh-CN,zh;q=0.8
```

第一部分：请求行，用来说明请求类型,要访问的资源以及所使用的 HTTP 版本. GET 说明请求类型为 GET,[/562f25980001b1b106000338.jpg]为要访问的资源，该行的最后一部分说明使用的是 HTTP1.1 版本。 第二部分：请求头部，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息 从第二行起为请求头部，HOST 将指出请求的目的地.User-Agent,服务器端和客户端脚本都能访问它,它是浏览器类型检测逻辑的重要基础.该信息由你的浏览器来定义,并且在每个请求中自动发送等等 第三部分：空行，请求头部后面的空行是必须的 即使第四部分的请求数据为空，也必须有空行。 第四部分：请求数据也叫主体，可以添加任意的其他数据。 这个例子的请求数据为空。

POST 请求例子

```
POST / HTTP1.1

Host:www.wrox.com

User-Agent:Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022)
```

```
Content-Type:application/x-www-form-urlencoded

Content-Length:40

Connection: Keep-Alive



name=Professional%20Ajax&publisher=Wiley
```

第一部分：请求行，第一行明了是 post 请求，以及 http1.1 版本。

第二部分：请求头部，第二行至第六行。

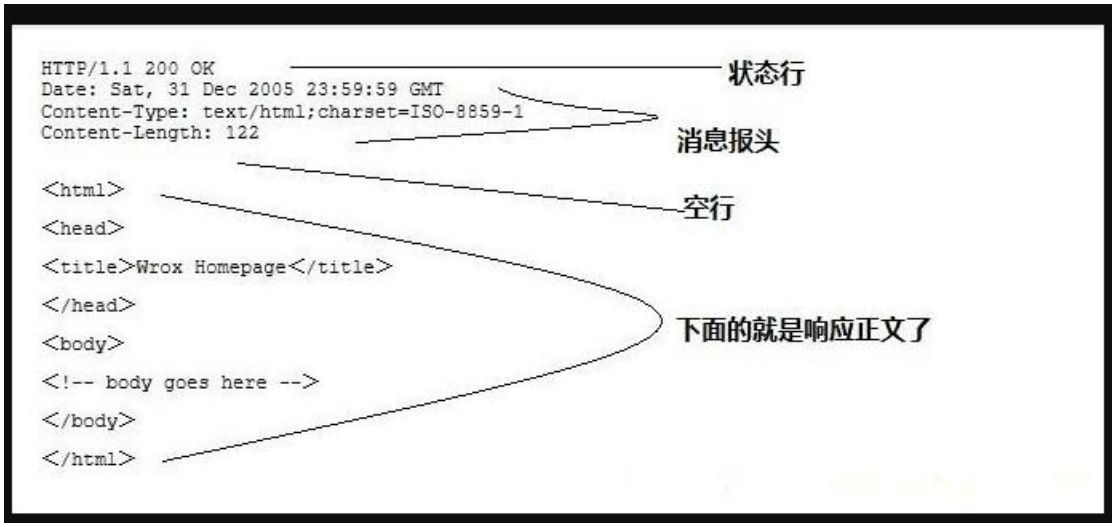
第三部分：空行，第七行的空行。

第四部分：请求数据，第八行。

2、Response 组成

一般情况下，服务器接收并处理客户端发过来的请求后会返回一个 HTTP 的响应消息。

HTTP 响应也由四个部分组成，分别是：状态行、消息报头、空行和响应正文。



第一部分：状态行，由 HTTP 协议版本号， 状态码， 状态消息 三部分组成。

第一行为状态行，（HTTP/1.1）表明 HTTP 版本为 1.1 版本，状态码为 200，状态消息为（ok）

第二部分：消息报头，用来说明客户端要使用的一些附加信息

第二行和第三行为消息报头， Date:生成响应的日期和时间；Content-Type:指定了 MIME 类型的 HTML(text/html),编码类型是 UTF-8

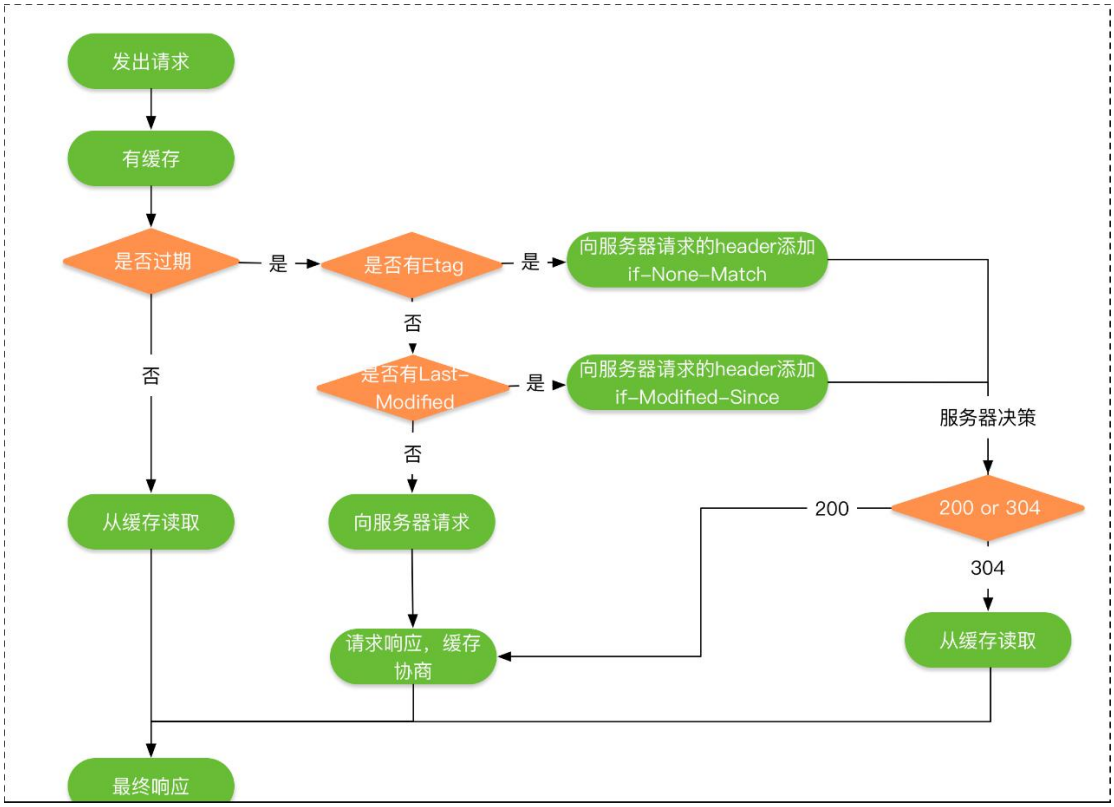
第三部分：空行，消息报头后面的空行是必须的

第四部分：响应正文，服务器返回给客户端的文本信息。

空行后面的 html 部分为响应正文。

5、谈谈对 http 缓存的了解。

HTTP 的缓存机制也是依赖于请求和响应 header 里的参数类实现的，最终响应式从缓存中去，还是从服务端重新拉取，HTTP 的缓存机制的流程如下所示：



HTTP 的缓存可以分为两种：

强制缓存：需要服务端参与判断是否继续使用缓存，当客户端第一次请求数据是，服务端返回了缓存的过期时间（Expires 与 Cache-Control），没有过期就可以继续使用缓存，否则则不适用，无需再向服务端询问。 对比缓存：需要服务端参与判断是否继续使用缓存，当客户端第一次请求数据时，服务端会将缓存标识（Last-Modified/If-Modified-Since 与 Etag/If-None-Match）与数据一起返回给客户端，客户端将两者都备份到缓存中 ，再次请求数据时，客户端将上次备份的缓存 标识发送给服务端，服务端根据缓存标识进行判断，如果返回 304，则表示通知客户端可以继续使用缓存。 强制缓存优先于对比缓存。

上面提到强制缓存使用的的两个标识：

Expires：Expires 的值为服务端返回的到期时间，即下一次请求时，请求时间小于服务端返回的到期时间，直接使用缓存数据。到期时间是服务端生成的，客户端和服务端的时间可能有误差。 Cache-Control：Expires 有个时间校验的问题，所有 HTTP1.1 采用 Cache-Control 替代 Expires。 Cache-Control 的取值有以下几种：

private: 客户端可以缓存。 public: 客户端和代理服务器都可缓存。 max-age=xxx: 缓存的内容将在 xxx 秒后失效 no-cache: 需要使用对比缓存来验证缓存数据。 no-store: 所有内容都不会缓存，强制缓存，对比缓存都不会触发。 我们再来看看对比缓存的两个标识：

Last-Modified/If-Modified-Since

Last-Modified 表示资源上次修改的时间。

当客户端发送第一次请求时，服务端返回资源上次修改的时间：

```
Last-Modified: Tue, 12 Jan 2016 09:31:27 GMT
```

客户端再次发送，会在 header 里携带 If-Modified-Since。将上次服务端返回的资源时间上传给服务端。

```
If-Modified-Since: Tue, 12 Jan 2016 09:31:27 GMT
```

服务端接收到客户端发来的资源修改时间，与自己当前的资源修改时间进行对比，如果自己的资源修改时间大于客户端发来的资源修改时间，则说明资源做过修改， 则返回 200 表示需要重新请求资源，否则返回 304 表示资源没有被修改，可以继续使用缓存。

上面是一种时间戳标记资源是否修改的方法，还有一种资源标识码 ETag 的方式来标记是否修改，如果标识码发生改变，则说明资源已经被修改，ETag 优先级高于 Last-Modified。

```
Etag/If-None-Match
```

ETag 是资源文件的一种标识码，当客户端发送第一次请求时，服务端会返回当前资源的标识码：

```
ETag: "5694c7ef-24dc"
```

客户端再次发送，会在 header 里携带上次服务端返回的资源标识码：

If-None-Match:"5694c7ef-24dc" 服务端接收到客户端发来的资源标识码，则会与自己当前的资源吗进行比较，如果不同，则说明资源已经被修改，则返回 200，如果相同则说明资源没有被修改，返回 304，客户端可以继续使用缓存。

6、Http 长连接。

Http1.0 是短连接，HTTP1.1 默认是长连接，也就是默认 Connection 的值就是 keep-alive。但是长连接实质是指的 TCP 连接，而不是 HTTP 连接。TCP 连接是一个双向的通道，它是可以保持一段时间不关闭的，因此 TCP 连接才有真正的长连接和短连接这一说。

Http1.1 为什么要用使用 tcp 长连接？

长连接是指的 TCP 连接，也就是说复用的是 TCP 连接。即长连接情况下，多个 HTTP 请求可以复用同一个 TCP 连接，这就节省了很多 TCP 连接建立和断开的消耗。

此外，长连接并不是永久连接的。如果一段时间内（具体的时间长短，是可以在 header 当中进行设置的，也就是所谓的超时时间），这个连接没有 HTTP 请求发出的话，那么这个长连接就会被断掉。

7、Https 加密原理。

加密算法的类型基本上分为了两种：

- 对称加密，加密用的密钥和解密用的密钥是同一个，比较有代表性的就是 AES 加密算法；
- 非对称加密，加密用的密钥称为公钥，解密用的密钥称为私钥，经常使用到的 RSA 加密算法就是非对称加密的；

此外，还有 Hash 加密算法

HASH 算法：MD5, SHA1, SHA256

相比较对称加密而言，非对称加密安全性更高，但是加解密耗费的时间更长，速度慢。

HTTPS = HTTP + SSL，HTTPS 的加密就是在 SSL 中完成的。

这就要从 CA 证书讲起了。CA 证书其实就是数字证书，是由 CA 机构颁发的。至于 CA 机构的权威性，那么是毋庸置疑的，所有人都是信任它的。CA 证书内一般会包含以下内容：

- 证书的颁发机构、版本
- 证书的使用者
- 证书的公钥
- 证书的有效时间
- 证书的数字签名 Hash 值和签名 Hash 算法
- ...

客户端如何校验 CA 证书？

CA 证书中的 Hash 值，其实是用证书的私钥进行加密后的值（证书的私钥不在 CA 证书中）。然后客户端得到证书后，利用证书中的公钥去解密该 Hash 值，得到 Hash-a ；然后再利用证书内的签名 Hash 算法去生成一个 Hash-b 。最后比较 Hash-a 和 Hash-b 这两个的值。如果相等，那么证明了该证书是对的，服务端是可以被信任的；如果不相等，那么就说明该证书是错误的，可能被篡改了，浏览器会给出相关提示，无法建立起 HTTPS 连接。除此之外，还会校验 CA 证书的有效时间和域名匹配等。

HTTPS 中的 SSL 握手建立过程

假设现在有客户端 A 和服务器 B ：

- 1、首先，客户端 A 访问服务器 B ，比如我们用浏览器打开一个网页 www.baidu.com ，这时，浏览器就是客户端 A ，百度的服务器就是服务器 B 了。这时候客户端 A 会生成一个随机数 1，把随机数 1 、自己支持的 SSL 版本号以及加密算法等这些信息告诉服务器 B 。
- 2、服务器 B 知道这些信息后，然后确认一下双方的加密算法，然后服务端也生成一个随机数 B ，并将随机数 B 和 CA 颁发给自己的证书一同返回给客户端 A 。
- 3、客户端 A 得到 CA 证书后，会去校验该 CA 证书的有效性，校验方法在上面已经说过了。校验通过后，客户端生成一个随机数 3 ，然后用证书中的公钥加密随机数 3 并传输给服务端 B 。
- 4、服务端 B 得到加密后的随机数 3，然后利用私钥进行解密，得到真正的随机数 3。
- 5、最后，客户端 A 和服务端 B 都有随机数 1、随机数 2、随机数 3，然后双方利用这三个随机数生成一个对话密钥。之后传输内容就是利用对话密钥来进行加解密了。这时就是利用了对称加密，一般用的都是 AES 算法。
- 6、客户端 A 通知服务端 B ，指明后面的通讯用对话密钥来完成，同时通知服务器 B 客户端 A 的握手过程结束。
- 7、服务端 B 通知客户端 A，指明后面的通讯用对话密钥来完成，同时通知客户端 A 服务器 B 的握手过程结束。
- 8、SSL 的握手部分结束，SSL 安全通道的数据通讯开始，客户端 A 和服务器 B 开始使用相同的对话密钥进行数据通讯。

简化如下：

- 1、客户端和服务端建立 SSL 握手，客户端通过 CA 证书来确认服务端的身份；
- 2、互相传递三个随机数，之后通过这随机数来生成一个密钥；
- 3、互相确认密钥，然后握手结束；
- 4、数据通讯开始，都使用同一个对话密钥来加解密；

可以发现，在 HTTPS 加密原理的过程中把对称加密和非对称加密都利用了起来。即利用了非对称加密安全性高的特点，又利用了对称加密速度快，效率高的好处。

8、HTTPS 如何防范中间人攻击？

什么是中间人攻击？

当数据传输发生在一个设备（PC/手机）和网络服务器之间时，攻击者使用其技能和工具将自己置于两个端点之间并截获数据；尽管交谈的两方认为他们是在与对方交谈，但是实际上他们是在与干坏事的人交流，这便是中间人攻击。

有几种攻击方式？

1、嗅探：嗅探或数据包嗅探是一种用于捕获流进和流出系统/网络的数据包的技术。网络中的数据包嗅探就好像电话中的监听。

2、数据包注入： 在这种技术中，攻击者会将恶意数据包注入常规数据中。这样用户便不会注意到文件/恶意软件，因为它们是合法通讯流的一部分。

3、会话劫持： 在你登录进你的银行账户和退出登录这一段期间便称为一个会话。这些会话通常都是黑客的攻击目标，因为它们包含潜在的重要信息。在大多数案例中，黑客会潜伏在会话中，并最终控制它。

4、SSL 剥离： 在 SSL 剥离攻击中，攻击者使 SSL/TLS 连接剥落，随之协议便从安全的 HTTPS 变成了不安全的 HTTP。

HTTPS 如何防范中间人攻击：

请见 https 加密原理。

9、有哪些响应码，分别都代表什么意思？

1** 信息，服务器收到请求，需要请求者继续执行操作

2** 成功，操作被成功接收并处理

3** 重定向，需要进一步的操作以完成请求

4** 客户端错误，请求包含语法错误或无法完成请求

5** 服务器错误，服务器在处理请求的过程中发生了错误

二、TCP/UDP （★ ★ ★）

1、为什么 tcp 要经过三次握手，四次挥手？

重要标志位

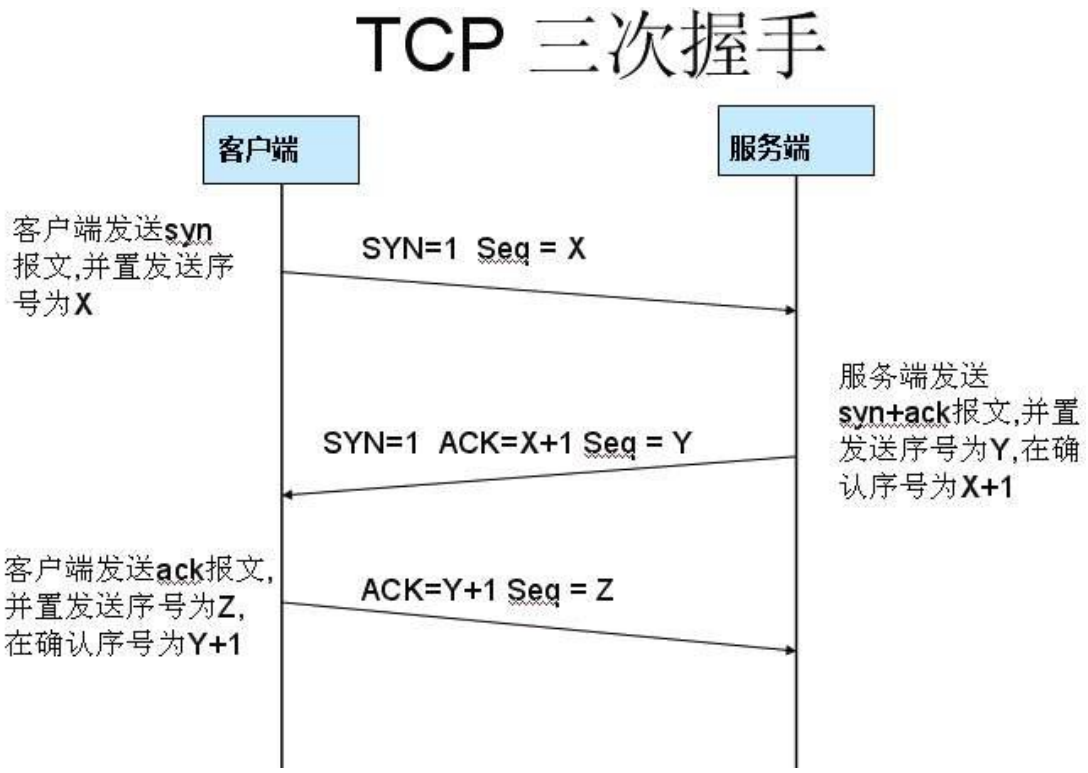
ACK ： TCP 协议规定，只有 ACK=1 时有效，也规定连接建立后所有发送的报文的 ACK 必须为 1

SYN(SYNchronization) ： 在连接建立时用来同步序号。当 SYN=1 而 ACK=0 时，表明这是一个连接请求报文。对方若同意建立连接，则应在响应报文中使 SYN=1 和 ACK=1. 因此, SYN 置 1 就表示这是一个连接请求或连接接受报文。

FIN （finis）即完，终结的意思， 用来释放一个连接。当 FIN = 1 时，表明此报文段的发送方的数据已经发送完毕，并要求释放连接。

三次握手、四次挥手过程

三次握手：

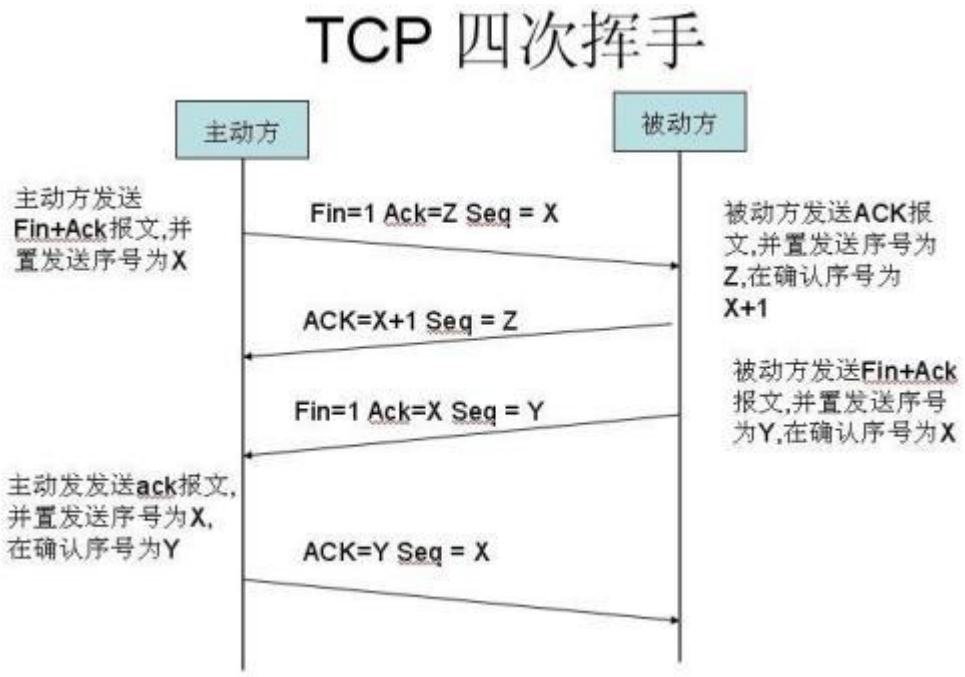


第一次握手：建立连接。客户端发送连接请求报文段，将 SYN 位置为 1，Sequence Number 为 x；然后，客户端进入 SYN_SEND 状态，等待服务器的确认；

第二次握手：服务器收到 SYN 报文段。服务器收到客户端的 SYN 报文段，需要对这个 SYN 报文段进行确认，设置 Acknowledgment Number 为 x+1(Sequence Number+1)；同时，自己自己还要发送 SYN 请求信息，将 SYN 位置为 1，Sequence Number 为 y；服务器端将上述所有信息放到一个报文段（即 SYN+ACK 报文段）中，一并发送给客户端，此时服务器进入 SYN_RECV 状态；

第三次握手：客户端收到服务器的 SYN+ACK 报文段。然后将 Acknowledgment Number 设置为 y+1，向服务器发送 ACK 报文段，这个报文段发送完毕以后，客户端和服务器端都进入 ESTABLISHED 状态，完成 TCP 三次握手。

四次挥手：



第一次分手：主机 1（可以使客户端，也可以是服务器端），设置 Sequence Number 和 Acknowledgment Number，向主机 2 发送一个 FIN 报文段；此时，主机 1 进入 FIN_WAIT_1 状态；这表示主机 1 没有数据要发送给主机 2 了；

第二次分手：主机 2 收到了主机 1 发送的 FIN 报文段，向主机 1 回一个 ACK 报文段，Acknowledgment Number 为 Sequence Number 加 1；主机 1 进入 FIN_WAIT_2 状态；主机 2 告诉主机 1，我"同意"你的关闭请求；

第三次分手：主机 2 向主机 1 发送 FIN 报文段，请求关闭连接，同时主机 2 进入 LAST_ACK 状态；

第四次分手：主机 1 收到主机 2 发送的 FIN 报文段，向主机 2 发送 ACK 报文段，然后主机 1 进入 TIME_WAIT 状态；主机 2 收到主机 1 的 ACK 报文段以后，就关闭连接；此时，主机 1 等待 2MSL 后依然没有收到回复，则证明 Server 端已正常关闭，那好，主机 1 也可以关闭连接了。

"三次握手"的目的是"为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误"。主要目的防止 server 端一直等待，浪费资源。换句话说，即是为了保证服务端能收接受到客户端的信息并能做出正确的应答而进行前两次(第一次和第二次)握手，为了保证客户端能够接收到服务端的信息并能做出正确的应答而进行后两次(第二次和第三次)握手。

"四次挥手"原因是因为 tcp 是全双工模式，接收到 FIN 时意味将没有数据再发来，但是还是可以继续发送数据。

2、TCP 可靠传输原理实现（滑动窗口）。

确认和重传：接收方收到报文后就会进行确认，发送方一段时间没有收到确认就会重传。

数据校验。

数据合理分片与排序，TCP 会对数据进行分片，接收方会缓存为按序到达的数据，重新排序后再提交给应用层。

流程控制：当接收方来不及接收发送的数据时，则会提示发送方降低发送的速度，防止包丢失。

拥塞控制：当网络发生拥塞时，减少数据的发送。

3、Tcp 和 Udp 的区别？

1、基于连接与无连接；

2、对系统资源的要求（TCP 较多，UDP 少）；

3、UDP 程序结构较简单；

4、流模式与数据报模式 ；

5、TCP 保证数据正确性，UDP 可能丢包；

6、TCP 保证数据顺序，UDP 不保证。

4、如何设计在 UDP 上层保证 UDP 的可靠性传输？

传输层无法保证数据的可靠传输，只能通过应用层来实现了。实现的方式可以参照 tcp 可靠性传输的方式。如不考虑拥塞处理，可靠 UDP 的简单设计如下：

- 1、添加 seq/ack 机制，确保数据发送到对端
- 2、添加发送和接收缓冲区，主要是用户超时重传。
- 3、添加超时重传机制。

具体过程即是：送端发送数据时，生成一个随机 seq=x，然后每一片按照数据大小分配 seq。数据到达接收端后接收端放入缓存，并发送一个 ack=x 的包，表示对方已经收到了数据。发送端收到了 ack 包后，删除缓冲区对应的数据。时间到后，定时任务检查是否需要重传数据。

目前有如下开源程序利用 udp 实现了可靠的数据传输。分别为 RUDP、RTP、UDT:

1、RUDP（Reliable User Datagram Protocol）

RUDP 提供一组数据服务质量增强机制，如拥塞控制的改进、重发机制及淡化服务器算法等。

2、RTP（Real Time Protocol）

RTP 为数据提供了具有实时特征的端对端传送服务，如在组播或单播网络服务下的交互式视频音频或模拟数据。

3、UDT（UDP-based Data Transfer Protocol）

UDT 的主要目的是支持高速广域网上的海量数据传输。

三、其它重要网络概念 （★★）

1、socket 断线重连怎么实现，心跳机制又是怎样实现？

socket 概念

套接字（socket）是通信的基石，是支持 TCP/IP 协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的 IP 地址，本地进程的协议端口，远地主机的 IP 地址，远地进程的协议端口。

为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与 TCP / IP 协议交互提供了套接字(Socket)接口。应 用层可以和传输层通过 Socket 接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

建立 socket 连接

建立 Socket 连接至少需要一对套接字，其中一个运行于客户端，称为 ClientSocket ，另一个运行于服务器端，称为 ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

- 服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求。
- 客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端- -套接字的地址和端口号，然后就向服务器端套接字提出连接请求。 连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发 给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

SOCKET 连接与 TCP 连接

创建 Socket 连接时，可以指定使用的传输层协议，Socket 可以支持不同的传输层协议（TCP 或 UDP），当使用 TCP 协议进行连接时，该 Socket 连接就是一个 TCP 连接。

Socket 连接与 HTTP 连接

由于通常情况下 Socket 连接就是 TCP 连接，因此 Socket 连接一旦建立，通信双方即可开始相互发送数据内容，直到双方连接断开。但在实际网 络应用中，客户端到服务器之间的通信往往需要穿越多个中间节点，例如路由器、网关、防火墙等，大部分防火墙默认会关闭长时间处于非活跃状态的连接而导致 Socket 连接断连，因此需要通过轮询告诉网络，该连接处于活跃状态。

而 HTTP 连接使用的是"请求—响应"的方式，不仅在请求时需要先建立连接，而且需要客户端向服务器发出请求后，服务器端才能回复数据。

很多情况下，需要服务器端主动向客户端推送数据，保持客户端与服务器数据的实时与同步。此时若双方建立的是 Socket 连接，服务器就可以直接将数 据传送给客户端；若双方建立的是 HTTP 连接，则服务器需要等到客户端发送一次请求后才能将数据传回给客户端，因此，客户端定时向服务器端发送连接请求， 不仅可以保持在线，同时也是在"询问"服务器是否有新的数据，如果有就将数据传给客户端。TCP(Transmission Control Protocol) 传输控制协议

socket 断线重连实现

正常连接断开客户端会给服务端发送一个 fin 包，服务端收到 fin 包后才会知道连接断开。而断网断电时客户端无法发送 fin 包给服务端，所以服务端没办法检测到客户端已经短线。 为了缓解这个问题，服务端需要有个心跳逻辑，就是服务端检测到某个客户端多久没发送任何数据过来就认为客户端已经断开， 这需要客户端定时向服务端发送心跳数据维持连接。

心跳机制实现

长连接的实现：心跳机制，应用层协议大多都有 HeartBeat 机制，通常是客户端每隔一小段时间向服务器发送一个数据包，通知服务器自己仍然在线。并传输一些可能必要的数据。使用心跳包的典型协议是 IM，比如 QQ/MSN/飞信等协议

1、在 TCP 的机制里面，本身是存在有心跳包的机制的，也就是 TCP 的选项：SO_KEEPALIVE。系统默认是设置的 2 小时的心跳频率。但是它检查不到机器断电、网线拔出、防火墙这些断线。 而且逻辑层处理断线可能也不是那么好处理。一般，如果只是用于保活还是可以的。通过使用 TCP 的 KeepAlive 机制（修改那个 time 参数），可以让连接每隔一小段时间就产生一些 ack 包，以降低被踢掉的风险，当然，这样的代价是额外的网络和 CPU 负担。

2、应用层心跳机制实现。

2、Cookie 与 Session 的作用和原理。

- Session 是在服务端保存的一个数据结构，用来跟踪用户的状态，这个数据可以保存在集群、数据库、文件中。
- Cookie 是客户端保存用户信息的一种机制，用来记录用户的一些信息，也是实现 Session 的一种方式。

Session：

由于 HTTP 协议是无状态的协议，所以服务端需要记录用户的状态时，就需要用某种机制来识具体的用户，这个机制就是 Session.典型的场景比如购物车，当你点击下单按钮时，由于 HTTP 协议无状态，所以并不知道是哪个用户操作的，所以服务端要为特定的用户创建了特定的 Session，用用于标识这个用户，并且跟踪用户，这样才知道购物车里面有几本书。这个 Session 是保存在服务端的，有一个唯一标识。在服务端保存 Session 的方法很多，内存、数据库、文件都有。集群的时候也要考虑 Session 的转移，在大型的网站，一般会有专门的 Session 服务器集群，用来保存用户会话，这个时候 Session 信息都是放在内存的。

具体到 Web 中的 Session 指的就是用户在浏览某个网站时，从进入网站到浏览器关闭所经过的这段时间，也就是用户浏览这个网站所花费的时间。因此从上述的定义中我们可以看到，Session 实际上是一个特定的时间概念。

当客户端访问服务器时，服务器根据需求设置 Session，将会话信息保存在服务器上，同时将标示 Session 的 SessionId 传递给客户端浏览器，

浏览器将这个 SessionId 保存在内存中，我们称之为无过期时间的 Cookie。浏览器关闭后，这个 Cookie 就会被清掉，它不会存在于用户的 Cookie 临时文件。

以后浏览器每次请求都会额外加上这个参数值，服务器会根据这个 SessionId，就能取得客户端的数据信息。

如果客户端浏览器意外关闭，服务器保存的 Session 数据不是立即释放，此时数据还会存在，只要我们知道那个 SessionId,就可以继续通过请求获得此 Session 的信息，因为此时后台的 Session 还存在，当然我们可以设置一个 Session 超时时间，一旦超过规定时间没有客户端请求时，服务器就会清除对应 SessionId 的 Session 信息。

Cookie

Cookie 是由服务器端生成，发送给 User-Agent（一般是 web 浏览器），浏览器会将 Cookie 的 key/value 保存到某个目录下的文本文件内，下次请求同一网站时就发送该 Cookie 给服务器（前提是浏览器设置为启用 Cookie）。Cookie 名称和值可以由服务器端开发自己定义，对于 JSP 而言也可以直接写入 Sessionid，这样服务器可以知道该用户是否合法用户以及是否需要重新登录等。

3、IP 报文中的内容。

版本：IP 协议的版本，目前的 IP 协议版本号为 4，下一代 IP 协议版本号为 6。

首部长度：IP 报头的长度。固定部分的长度（20 字节）和可变部分的长度之和。共占 4 位。最大为 1111，即 10 进制的 15，代表 IP 报头的最大长度可以为 15 个 32bits（4 字节），也就是最长可为 15*4=60 字节，除去固定部分的长度 20 字节，可变部分的长度最大为 40 字节。

服务类型：Type Of Service。

总长度：IP 报文的总长度。报头的长度和数据部分的长度之和。

标识：唯一的标识主机发送的每一分数据报。通常每发送一个报文，它的值加一。当 IP 报文长度超过传输网络的 MTU（最大传输单元）时必须分片，这个标识字段的值被复制到所有数据分片的标识字段中，使得这些分片在达到最终目的地时可以依照标识字段的内容重新组成原先的数据。

标志：共 3 位。R、DF、MF 三位。目前只有后两位有效，DF 位：为 1 表示不分片，为 0 表示分片。MF：为 1 表示“更多的片”，为 0 表示这是最后一片。

片位移：本分片在原先数据报文中相对首位的偏移位。（需要再乘以 8）

生存时间：IP 报文所允许通过的路由器的最大数量。每经过一个路由器，TTL 减 1，当为 0 时，路由器将该数据报丢弃。TTL 字段是由发送端初始设置一个 8 bit 字段.推荐的初始值由分配数字 RFC 指定，当前值为 64。发送 ICMP 回显应答时经常把 TTL 设为最大值 255。

协议：指出 IP 报文携带的数据使用的是那种协议，以便目的主机的 IP 层能知道要将数据报上交到哪个进程（不同的协议有专门不同的进程处理）。和端口号类似，此处采用协议号，TCP 的协议号为 6，UDP 的协议号为 17。ICMP 的协议号为 1，IGMP 的协议号为 2.

首部校验和：计算 IP 头部的校验和，检查 IP 报头的完整性。

源 IP 地址：标识 IP 数据报的源端设备。

目的 IP 地址：标识 IP 数据报的目的地址。

最后就是可变部分和数据部分。

四、常见网络流程机制 （★ ★ ）

1、浏览器输入地址到返回结果发生了什么？

总体来说分为以下几个过程:

1、DNS 解析，此外还有 DNSy 优化（DNS 缓存、DNS 负载均衡）

2、TCP 连接

3、发送 HTTP 请求

4、服务器处理请求并返回 HTTP 报文

5、浏览器解析渲染页面

6、连接结束

Web 前端的本质

将信息快速并友好的展示给用户并能够与用户进行交互。

如何尽快的加载资源（网络优化）？

答案就是能不从网络中加载的资源就不从网络中加载，当我们合理使用缓存，将资源放在浏览器端，这是最快的方式。如果资源必须从网络中加载，则要考虑缩短连接时间，即 DNS 优化部分;减少响应内容大小，即对内容进行压缩。另一方面，如果加载的资源数比较少的话，也可以快速的响应用户。

第二节、操作系统面试题 （★ ★ ★ ）

1、操作系统如何管理内存的？

2、进程调度。

3、说下 Linux 进程和线程的区别。

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

> 简而言之,一个程序至少有一个进程,一个进程至少有一个线程。
>
> 线程的划分尺度小于进程，使得多线程程序的并发性高。
>
> 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
>
> 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
>
> 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

4、你能解释一下 Linux 的软链接和硬链接吗？

Linux 链接分两种，一种被称为硬链接（Hard Link），另一种被称为符号链接（Symbolic Link）。默认情况下，ln 命令产生硬链接。

硬连接

硬连接指通过索引节点来进行连接。在 Linux 的文件系统中，保存在磁盘分区中的文件不管是什么类型都给它分配一个编号，称为索引节点号(Inode Index)。在 Linux 中，多个文件名指向同一索引节点是存在的。一般这种连接就是硬连接。硬连接的作用是允许一个文件拥有多个有效路径名，这样用户就可以建立硬连接

到重要文件，以防止“误删”的功能。其原因如上所述，因为对应该目录的索引节点有一个以上的连接。只删除一个连接并不影响索引节点本身和其它的连接，只有当最后一个连接被删除后，文件的数据块及目录的连接才会被释放。也就是说，文件真正删除的条件是与之相关的所有硬连接文件均被删除。

软连接

另外一种连接称之为符号连接（Symbolic Link），也叫软连接。软链接文件有类似于 Windows 的快捷方式。它实际上是一个特殊的文件。在符号连接中，文件实际上是一个文本文件，其中包含的有另一文件的位置信息。

5、安卓权限管理，为何在清单中注册权限，安卓 APP 就可以使用，反之不可以？

此题考查 Android 的权限管理在 Android 的安全架构中的作用。

Android 是一个权限分隔的操作系统，其中每个应用都有其独特的系统标识（Linux 用户 ID 和组 ID）。系统各部分也分隔为不同的标识。Linux 据此将不同的应用以及应用与系统分隔开来。

其他更详细的安全功能通过“权限”机制提供，此机制会限制特定进程可以执行的具体操作，并且根据 URI 权限授权临时访问特定的数据段。

Android 安全架构的中心设计点是：在默认情况下任何应用都没有权限执行对其他应用、操作系统或用户有不利影响的任何操作。这包括读取或写入用户的私有数据（例如联系人或电子邮件）、读取或写入其他应用程序的文件、执行网络访问、使设备保持唤醒状态等。

由于每个 Android 应用都是在进程沙盒中运行，因此应用必须显式共享资源和数据。它们的方法是声明需要哪些权限来获取基本沙盒未提供的额外功能。应用以静态方式声明它们需要的权限，然后 Android 系统提示用户同意。

第三节、数据库面试题 （★）

1、数据库的四大特征，数据库的隔离级别？

事务（Transaction）是并发控制的基本单位。所谓的事务，它是一个操作序列，这些操作要么都执行，要么都不执行，它是一个不可分割的工作单位。例如，银行转账工作：从一个账号扣款并使另一个账号增款，这两个操作要么都执行，要么都不执行。所以，应该把它们看成一个事务。事务是数据库维护数据一致性的单位，在每个事务结束时，都能保持数据一致性。事务具有以下 4 个基本特征：

数据库的四大特征：

（1）原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。

（2）一致性（Consistency）

一个事务执行之前和执行之后都必须处于一致性状态。

（3）隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

（4）持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的。

数据库的隔离级别：

1）Serializable(串行化)：可避免脏读、不可重复读、幻读的发生。

2）Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

3）Read committed (读已提交)：可避免脏读的发生。

4）Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

2、数据库设计中常讲的三范式是指什么？

1）第一范式 1NF(域的原子性)

如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式

2）第二范式 2NF(表中除主键外的字段都完全依赖主键)

第二范式是在第一范式基础上建立的。第二范式有两个重点:(1)表中必须有主键；(2)其他非主属性必须完全依赖主键，不能只依赖主键的一部分（主要针对联合主键而言）。

3）第三范式 3NF（表中除主键外的字段都完全直接依赖，不能是传递依赖）

不能是传递依赖，即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。第二范式和第三范式区分的关键点：2NF：非主键列是否完全依赖于主键，还是依赖于主键的一部分；3NF：非主键列是直接依赖于主键，还是直接依赖于非主键列。

第二章 数据结构和算法面试题

数据结构与算法

对于算法面试准备，无疑就是刷《剑指 Offer》+ LeetCode 效果最佳。刷《剑指 Offer》是为了建立全面的算法面试思维，打下坚实的基础，刷 LeetCode 则是为了不断强化与开阔我们自己的算法思想。这两块 **CS-Notes** 中已经实现地很

完美了，**建议大家将《剑指 Offer》刷完，然后再至少刷 100 道 LeetCode 题目以上。**

1、剑指 Offer 题解

2、Leetcode 题解

> 建议刷完《剑指 Offer》+ 至少 100 道 LeetCode 题目后，再去进一步看看下面的高频题目有哪些没有刷到。

一、高频题集 （★ ★ ★）

1、无重复字符的最长子串

2、简化路径

3、复原 IP 地址

4、三数之和

5、岛屿的最大面积

6、搜索旋转排序数组

7、朋友圈

8、接雨水

9、反转链表

10、两数相加

11、合并两个有序链表

12、合并 K 个排序链表

13、买卖股票的最佳时机

14、买卖股票的最佳时机 II

15、最大子序和

16、最小栈

17、LRU 缓存机制

18、寻找两个有序数组的中位数

19、最长回文子串

20、合并两个有序数组

21、整数反转

22、排序链表

23、子集

24、全排列

25、实现二叉树中序遍历（不使用递归）

26、爬楼梯（斐波那契数列）

27、滑动窗口的最大值

28、判断单链表成环与否？

29、如何从一百万个数里面找到最小的一百个数，考虑算法的时间复杂度和空间复杂度

30、手写数组实现队列

31、java 排序算法和查找算法（写出你所知道的排序算法及时空复杂度，稳定性）

http://www.jianshu.com/p/8c915179fd02

http://xiaojun-it.iteye.com/blog/2291852

二、次高频题集 （★ ★ ）

1、算法熟悉么？给了一个二叉排序树，出了一个给定节点找到它的下一个元素（指的是大小顺序的下一个）的算法题。

2、x 个苹果，一天只能吃一个、两个、或者三个，问多少天可以吃完

3、求二叉树第 n 层节点数

4、如何设计一个抽奖系统，比如满 200 抽 20，满 500 抽 50。

5、求无序数组中的中位数

6、二叉树深度算法

7、堆和栈在内存中的区别是什么(数据结构方面以及实际实现方面)

8、最快的排序算法是哪个？给阿里 2 万多名员工按年龄排序应该选择哪个算法？

9、堆和树的区别？

10、求 1000 以内的水仙花数以及 40 亿以内的水仙花数；

11、子串包含问题(KMP 算法)写代码实现；

12、万亿级别的两个 URL 文件 A 和 B，如何求出 A 和 B 的差集 C,(Bit 映射->hash 分组->多文件读写效率->磁盘寻址以及应用层面对寻址的优化)

13、蚁群算法与蒙特卡洛算法；

14、百度 POI 中如何试下查找最近的商家功能(坐标镜像+R 树)。

15、5 枚硬币，2 正 3 反如何划分为两堆然后通过翻转让两堆中正面向上的硬币和反面向上的硬币个数相同；

16、时针走一圈，时针分针重合几次；

17、N * N 的方格纸,里面有多少个正方形；

18、请在 100 个电话号码找出 135 的电话号码 注意 不能用正则，(类似怎么最好的遍历 LogGat 日志）

19、一个青蛙跳台阶，一次可以跳一步和两步，如果一共有 N 个台阶，可以有几种跳法？

20、写代码实现队列的基本操作，外加查找最大值；

21、图：有向无环图的解释

22、二叉树 深度遍历与广度遍历

23、B 树、B+树

24、密码学中两大加密算法是什么

25、判断环（猜测应该是链表环）

26、有一个 List 列表，去掉列表中的某一 Object 对象，如何在 for 循环里面写；

27、设计移动端的联系人存储与查询的功能，要求快速搜索联系人，可以用到哪些数据结构？（二叉排序树，建立索引）

28、一道简单不易的算法题

```
    int  a = 10;

    int b=5;

    怎么在不引入其他变量的情况下,让 a 和 b 互换？


```

```
public class Test {

   int  a = 10;

   int b=5;

public static void main(String[] args) {

   a = a+b;

   b=a-b;

   a =a-b;

   System.out.println("b="+b);

   System.out.println("a="+a);

 }

}

----输出：

b=10

a=5
```

29、回形打印二维数组

30、二叉树，给出根节点和目标节点，找出从根节点到目标节点的路径

31、一个无序，不重复数组，输出 N 个元素，使得 N 个元素的和相加为 M，给出时间复杂度、空间复杂度。手写算法

32、两个不重复的数组集合中，求共同的元素。

33、上一问扩展，海量数据，内存中放不下，怎么求出。

34、从长度为 m 的 int 数组中随机取出 n 个元素，每次取的元素都是之前未取过的，如何优化

35、逆序一个字符串，不能调用 String 的 reverse 方法(考察编码风格)

36、算法：将一个有序数组去重得到一个新数组(空间复杂度为 O(N))

37、算法：如何从 1T 的无序数组(长度为 n)里面找出前 k 大的数据，复杂度要求为 O(logN)

38、m * n 的矩阵，能形成几个正方形（2 * 2 能形成 1 个正方形，2 * 3 2 个，3 * 3 6 个）

计数的关键是要观察到任意一个倾斜的正方形必然唯一内接于一个非倾斜的正方形，而一个非倾斜的边长为 k 的非倾斜正方形，一条边上有 k-1 个内点，每个内点恰好确定一个内接于其中的倾斜正方形，加上非倾斜正方形本身，可知，将边长为 k 的非倾斜正方形数目乘以 k，再按 k 求和即可得到所有正方形的数目。

设 2≤n≤m，k≤n-1，则边长为 k 的非倾斜有

(n-k)(m-k)个，故所有正方形有

∑(m-k)(n-k)k 个

例如 m=n=4

正方形有 331+222+113=20 个。

39、面试头条的时候在线编程：从上到下从左到右输出二叉树

40、从长度为 m 的 int 数组中随机取出 n 个元素，每次取的元素都是之前未取过的，如何优化

41、逆序一个字符串，不能调用 String 的 reverse 方法(考察编码风格)

42、算法：将一个有序数组去重得到一个新数组(空间复杂度为 O(N))

43、算法：如何从 1T 的无序数组(长度为 n)里面找出前 k 大的数据，复杂度要求为 O(logN)

44、堆和树的区别

45、堆和栈在内存中的区别是什么(解答提示：可以从数据结构方面以及实际实现方面两个方面去回答)？

46、什么是深拷贝和浅拷贝

47、手写链表逆序代码

48、讲一下对图的理解

49、手写一段代码，如何找出一段字符串中，出现最多的汉字是哪个？

50、单向链表逆序。

51、实现一个数组插入。（处理异常判别，不使用 Collections 相关接口）。

52、找到无序数组的最大连续求和。

53、找到多个员工的共同繁忙时段。

https://github.com/banking/algorithm-dish/blob/master/algorithm-question/src/main/java/TimeAirbnb.java

54、输出一个集合{A,B,C,D}的全部子集**

55、手写代码：遍历文件目录；

56、电梯运行的算法分析；

57、手写实现单链表的 get 操作；

58、100 个数字排序怎么做？

59、一个集合里有 1000 万个随机元素，如何快速计算他们的和（我特喵的以为是考算法，想半天没有 O（n）以下的方案，结果他居然说多线程）

60、切饼问题：1 刀切 2 块，2 刀切 4 块，10 刀最多切几块。

61、追击问题：2 辆火车相向同时出发，一个从 A 点出发，速度是 30km/h,一个从 B 点出发，速度是 20km/h，A、B 之间的距离是 S，同时一只鸟也从 A 点出发，速度是 50km/h，鸟在两辆火车间折返飞行，问三者相遇时，鸟飞的总路程。

62、算法：给定一段已排好序的数列，数列中元素有可能是重复的，找出数列中目标值的最小索引，要求不能用递归，只能用循环。

63、一个文件中有 100 万个整数，由空格分开，在程序中判断用户输入的整数是否在此文件中。说出最优的方法

64、2000 万个整数，找出第五十大的数字？

65、烧一根不均匀的绳，从头烧到尾总共需要 1 个小时。现在有若干条材质相同的绳子，问如何用烧绳的方法来计时一个小时十五分钟呢？

66、求 1000 以内的水仙花数以及 40 亿以内的水仙花数

67、称重问题：10 个箱子，每个箱子里都有若干砖头，其中有一个箱子里每个砖头重 9 克，其他的箱子里的砖头每个重 10 克，给你一个秤，要求只能用一次称重找出砖头重 9 克的箱子。

68、时针走一圈，时针分针重合几次

69、N*N 的方格纸,里面有多少个正方形

70、标号 1-n 的 n 个人首尾相接，1 到 3 报数，报到 3 的退出，求最后一个人的标号

71、给定一个字符串，求第一个不重复的字符 abbcad -> c

72、上网站写代码，如下： 有一个容器类 ArrayList，保存整数类型的元素，现在要求编写一个帮助类，类内提供一个帮助函数，帮助函数的功能是删除 容器中<10 的元素。

73、写代码，LeetCode 上股票利益最大化问题

74、写代码，剑指 offer 上第一次只出现一次的字符

75、算法：连续子数组的最大和

76、子串包含问题(KMP 算法)写代码实现

77、ViewGroup 的层级深度，转换为二叉树的层级深度

78、String 字符串的数字相加

79、使用三个线程顺序打印有序的数组

80、给定一个有序的数组和目标数，找出与目标数最近接的 index，要求复杂度是 log(n)的时间复杂度

81、给定一个二叉树和一个目标数，在二叉树中是否存在一条路径的所有节点的和与目标数是相同的 case，并且打印。

82、二叉树，读取每一层最右边的节点

83、int 数组，除了一个数字外，其他数字都出现两次，找出这个只出现一次的数字

84、一个类，内部有一个链表的数据结构，实现 void add(Node n)和 void remove(int index)的函数

85、手写消费者生产者模型的代码

86、一个无序的 int 数组，给一个 target 数字，找出数组中两个数字相加为 target，并输出坐标

87、数组中存有 1-3 的三种数字,例如[1,2,3,1,2,2,1,3,3]，将其排序为[1,1,1,2,2,2,3,3,3]，要求时间复杂度，后续将内容变为一个对象，继续排序

88、"之"字形打印二叉树

89、1~100 盏灯，都是亮的，第一次将能被 1 整除的数的灯按下，变暗，第二次将能被 2 整除的数的等按下，变亮，第三次将能被 3 整除的数的等按下，变暗…第 100 次将能被 100 整除的数的灯按下，问，最后有多少盏灯是亮的。

90、实现一个 o(n) 复杂度的堆和最大数。

91、实现一个数组的窗口扫描算法。

92、识别一个字符串是否是 ipv4 地址。

93、o(n)复杂度实现偶数递增奇数递减单向链接排序。

第三章 Java 面试题

第一节 Java 基础面试题

一、面向对象 （★★★）

1、谈谈对 java 多态的理解？

多态是指父类的某个方法被子类重写时，可以产生自己的功能行为，同一个操作作用于不同对象，可以有不同的解释，产生不同的执行结果。

多态的三个必要条件：

- 继承父类。
- 重写父类的方法。
- 父类的引用指向子类对象。

什么是多态

面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间的耦合关系。

现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

多态的好处：

1.可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆 Circle 类工作，对其他任何圆形几何体，如圆环，也同样工作。

2.可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

3.接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。

4.灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。

5.简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

Java 中多态的实现方式：接口实现，继承父类进行方法重写，同一个类中进行方法重载。

2、你所知道的设计模式有哪些？

答：Java 中一般认为有 23 种设计模式，我们不需要所有的都会，但是其中常用的种设计模式应该去掌握。下面列出了所有的设计模式。要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共五种：

工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：

适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：

策略模式、模板方法模式、观者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

3、通过静态内部类实现单例模式有哪些优点？

1. 不用 synchronized ，节省时间。
2. 调用 getInstance() 的时候才会创建对象，不调用不创建，节省空间，这有点像传说中的懒汉式。

4、静态代理和动态代理的区别，什么场景使用？

静态代理与动态代理的区别在于代理类生成的时间不同，即根据程序运行前代理类是否已经存在，可以将代理分为静态代理和动态代理。如果需要对多个类进行代理，并且代理的功能都是一样的，用静态代理重复编写代理类就非常的麻烦，可以用动态代理动态的生成代理类。

```
// 为目标对象生成代理对象

public Object getProxyInstance() {

    return Proxy.newProxyInstance(target.getClass().getClassLoader(),
target.getClass().getInterfaces(),

            new InvocationHandler() {



                @Override

                public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {

                    System.out.println("开启事务");



                    // 执行目标对象方法

                    Object returnValue = method.invoke(target, args);



                    System.out.println("提交事务");

                    return null;

                }

            });

}
```

- 静态代理使用场景：四大组件同 AIDL 与 AMS 进行跨进程通信
- 动态代理使用场景：Retrofit 使用了动态代理极大地提升了扩展性和可维护性。

5、简单工厂、工厂方法、抽象工厂、Builder 模式的区别？

- 简单工厂模式：一个工厂方法创建不同类型的对象。
- 工厂方法模式：一个具体的工厂类负责创建一个具体对象类型。
- 抽象工厂模式：一个具体的工厂类负责创建一系列相关的对象。
- Builder 模式：对象的构建与表示分离，它更注重对象的创建过程。

6、装饰模式和代理模式有哪些区别 ？与桥接模式相比呢？

- 1、装饰模式是以客户端透明的方式扩展对象的功能，是继承关系的一个替代方案；而代理模式则是给一个对象提供一个代理对象，并由代理对象来控制对原有对象的引用。
- 2、装饰模式应该为所装饰的对象增强功能；代理模式对代理的对象施加控制，但不对对象本身的功能进行增加。
- 3、桥接模式的作用于代理、装饰截然不同，它主要是为了应对某个类族有多个变化维度导致子类类型急剧增多的场景。通过桥接模式将多个变化维度隔离开，使得它们可以独立地变化，最后通过组合使它们应对多维变化，减少子类的数量和复杂度。

7、外观模式和中介模式的区别？

外观模式重点是对外封装统一的高层接口，便于用户使用；而中介模式则是避免多个互相协作的对象直接引用，它们之间的交互通过一个中介对象进行，从而使得它们耦合松散，能够易于应对变化。

8、策略模式和状态模式的区别？

虽然两者的类型结构是一致的，但是它们的本质却是不一样的。策略模式重在整个算法的替换，也就是策略的替换，而状态模式则是通过状态来改变行为。

9、适配器模式，装饰者模式，外观模式的异同？

这三个模式的相同之处是，它们都作用于用户与真实被使用的类或系统之间，作一个中间层，起到了让用户间接地调用真实的类的作用。它们的不同之外在于，如上所述的应用场合不同和本质的思想不同。

代理与外观的主要区别在于，代理对象代表一个单一对象，而外观对象代表一个子系统，代理的客户对象无法直接访问对象，由代理提供单独的目标对象的访问，而通常外观对象提供对子系统各元件功能的简化的共同层次的调用接口。代理是一种原来对象的代表，其它需要与这个对象打交道的操作都是和这个代表交涉

的。而适配器则不需要虚构出一个代表者，只需要为应付特定使用目的，将原来的类进行一些组合。

外观与适配器都是对现存系统的封装。外观定义的新的接口，而适配器则是复用一个原有的接口，适配器是使两个已有的接口协同工作，而外观则是为现存系统提供一个更为方便的访问接口。如果硬要说外观是适配，那么适配器有用来适配对象的，而外观是用来适配整个子系统的。也就是说，外观所针对的对象的粒度更大。

代理模式提供与真实的类一致的接口，意在用代理类来处理真实的类，实现一些特定的服务或真实类的部分功能，Facade（外观）模式注重简化接口，Adapter（适配器）模式注重转换接口。

10、代码的坏味道：

1、代码重复：

代码重复几乎是最常见的异味了。他也是 Refactoring 的主要目标之一。代码重复往往来自于 copy-and-paste 的编程风格。

2、方法过长：

一个方法应当具有自我独立的意图，不要把几个意图放在一起。

3、类提供的功能太多：

把太多的责任交给了一个类，一个类应该仅提供一个单一的功能。

4、数据泥团：

某些数据通常像孩子一样成群玩耍：一起出现在很多类的成员变量中，一起出现在许多方法的参数中.....，这些数据或许应该自己独立形成对象。 比如以单例的形式对外提供自己的实例。

5、冗赘类：

一个干活不多的类。类的维护需要额外的开销，如果一个类承担了太少的责任，应当消除它。

6、需要太多注释：

经常觉得要写很多注释表示你的代码难以理解。如果这种感觉太多，表示你需要Refactoring。

11、是否能从 Android 中举几个例子说说用到了什么设计模式 ？

AlertDialog、Notification 源码中使用了 Bulider（建造者）模式完成参数的初始化：

在 AlertDialog 的 Builder 模式中并没有看到 Direcotr 角色的出现，其实在很多场景中，Android 并没有完全按照 GOF 的经典设计模式来实现，而是做了一些修改，使得这个模式更易于使用。这个的 AlertDialog.Builder 同时扮演了上下文中提到的 builder、ConcreteBuilder、Director 的角色，简化了 Builder 模式的设计。当模块比较稳定，不存在一些变化时，可以在经典模式实现的基础上做出一些精简，而不是照搬 GOF 上的经典实现，更不要生搬硬套，使程序失去架构之美。

定义：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。即将配置从目标类中隔离出来，避免过多的 setter 方法。

优点：

- 1、良好的封装性，使用建造者模式可以使客户端不必知道产品内部组成的细节。
- 2、建造者独立，容易扩展。

缺点：

- 会产生多余的 Builder 对象以及 Director 对象，消耗内存。

日常开发的 BaseActivity 抽象工厂模式：

定义：为创建一组相关或者是相互依赖的对象提供一个接口，而不需要指定它们的具体类。

主题切换的应用：

比如我们的应用中有两套主题，分别为亮色主题 LightTheme 和暗色主题 DarkTheme，这两种主题我们可以通过一个抽象的类或接口来定义，而在对应主题下我们又有各类不同的 UI 元素，比如 Button、TextView、Dialog、ActionBar 等，这些 UI 元素都会分别对应不同的主题，这些 UI 元素我们也可以通过抽象的类或接口定义，抽象的主题、具体的主题、抽象的 UI 元素和具体的 UI 元素之间的关系就是抽象工厂模式最好的体现。

优点：

- 分离接口与实现，面向接口编程，使其从具体的产品实现中解耦，同时基于接口与实现的分离，使抽象该工厂方法模式在切换产品类时更加灵活、容易。

缺点：

- 类文件的爆炸性增加。
- 新的产品类不易扩展。

Okhttp 内部使用了责任链模式来完成每个 Interceptor 拦截器的调用：

定义：使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

ViewGroup 事件传递的递归调用就类似一条责任链，一旦其寻找到责任者，那么将由责任者持有并消费掉该次事件，具体体现在 View 的 onTouchEvent 方法中返回值的设置，如果 onTouchEvent 返回 false，那么意味着当前 View 不会是该次事件的责任人，将不会对其持有；如果为 true 则相反，此时 View 会持有该事件并不再向下传递。

优点：

将请求者和处理者关系解耦，提供代码的灵活性。

缺点：

对链中请求处理者的遍历中，如果处理者太多，那么遍历必定会影响性能，特别是在一些递归调用中，要慎重。

RxJava 的观察者模式：

定义：定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

ListView/RecyclerView 的 Adapter 的 notifyDataSetChanged 方法、广播、事件总线机制。

观察者模式主要的作用就是对象解耦，将观察者与被观察者完全隔离，只依赖于 Observer 和 Observable 抽象。

优点：

- 观察者和被观察者之间是抽象耦合，应对业务变化。
- 增强系统灵活性、可扩展性。

缺点：

- 在 Java 中消息的通知默认是顺序执行，一个观察者卡顿，会影响整体的执行效率，在这种情况下，一般考虑采用异步的方式。

AIDL 代理模式：

定义：为其他对象提供一种代理以控制对这个对象的访问。

静态代理：代码运行前代理类的 class 编译文件就已经存在。

动态代理：通过反射动态地生成代理者的对象。代理谁将会在执行阶段决定。将原来代理类所做的工作由 InvocationHandler 来处理。

使用场景：

- 当无法或不想直接访问某个对象或访问某个对象存在困难时可以通过一个代理对象来间接访问，为了保证客户端使用的透明性，委托对象与代理对象需要实现相同的接口。

缺点：

- 对类的增加。

ListView/RecyclerView/GridView 的适配器模式：

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

使用场景：

- 接口不兼容。
- 想要建立一个可以重复使用的类。
- 需要一个统一的输出接口，而输入端的类型不可预知。

优点：

- 更好的复用性：复用现有的功能。
- 更好的扩展性：扩展现有的功能。

缺点：

- 过多地使用适配器，会让系统非常零乱，不易于整体把握。例如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果出现太多这种情况，无异于一场灾难。

Context/ContextImpl 外观模式：

要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行，门面模式提供一个高层次的接口，使得子系统更易于使用。

使用场景：

- 为一个复杂子系统提供一个简单接口。

优点：

- 对客户程序隐藏子系统细节，因而减少了客户对于子系统的耦合，能够拥抱变化。
- 外观类对子系统的接口封装，使得系统更易用使用。

缺点：

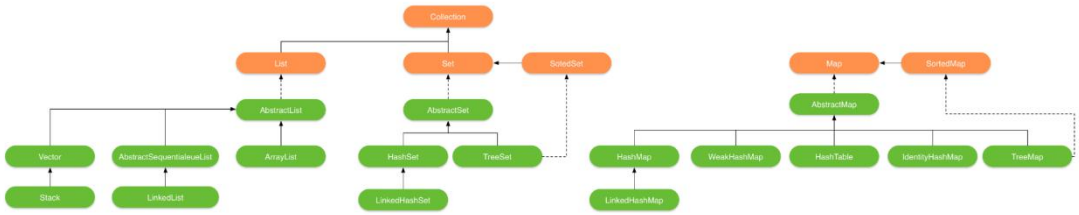
- 外观类接口膨胀。
- 外观类没有遵循开闭原则，当业务出现变更时，可能需要直接修改外观类。

二、集合框架 （★ ★ ★ ）

1、集合框架，list，map，set 都有哪些具体的实现类，区别都是什么？

Java 集合里使用接口来定义功能，是一套完善的继承体系。Iterator 是所有集合的总接口，其他所有接口都继承于它，该接口定义了集合的遍历操作，Collection 接口继承于 Iterator，是集合的次级接口（Map 独立存在），定义了集合的一些通用操作。

Java 集合的类结构图如下所示：



List：有序、可重复；索引查询速度快；插入、删除伴随数据移动，速度慢；

Set：无序，不可重复；

Map：键值对，键唯一，值多个；

1.List,Set 都是继承自 Collection 接口，Map 则不是;

2.List 特点：元素有放入顺序，元素可重复;

Set 特点：元素无放入顺序，元素不可重复，重复元素会盖掉，（注意：元素虽然无放入顺序，但是元素在 set 中位置是由该元素的 HashCode 决定的，其位置其实是固定，加入 Set 的 Object 必须定义 equals()方法;

另外 list 支持 for 循环，也就是通过下标来遍历，也可以使用迭代器，但是 set 只能用迭代，因为他无序，无法用下标取得想要的值）。

3.Set 和 List 对比：

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

List：和数组类似，List 可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

4.Map 适合储存键值对的数据。

5.线程安全集合类与非线程安全集合类

LinkedList、ArrayList、HashSet 是非线程安全的，Vector 是线程安全的;

HashMap 是非线程安全的，HashTable 是线程安全的;

StringBuilder 是非线程安全的，StringBuffer 是线程安的。

下面是这些类具体的使用介绍：

ArrayList 与 LinkedList 的区别和适用场景

Arraylist：

优点：ArrayList 是实现了基于动态数组的数据结构,因地址连续，一旦数据存储好了，查询操作效率会比较高（在内存里是连着放的）。

缺点：因为地址连续，ArrayList 要移动数据,所以插入和删除操作效率比较低。

LinkedList：

优点：LinkedList 基于链表的数据结构，地址是任意的，其在开辟内存空间的时候不需要等一个连续的地址，对新增和删除操作 add 和 remove，LinedList 比较占优势。LikedList 适用于要头尾操作或插入指定位置的场景。

缺点：因为 LinkedList 要移动指针,所以查询操作性能比较低。

适用场景分析：

当需要对数据进行对此访问的情况下选用 ArrayList，当要对数据进行多次增加删除修改时采用 LinkedList。

ArrayList 和 LinkedList 怎么动态扩容的吗？

ArrayList:

ArrayList 初始化大小是 10 （如果你知道你的 arrayList 会达到多少容量，可以在初始化的时候就指定，能节省扩容的性能开支） 扩容点规则是，新增的时候发现容量不够用了，就去扩容 扩容大小规则是，扩容后的大小= 原始大小+原始大小/2 + 1。（例如：原始大小是 10 ，扩容后的大小就是 10 + 5+1 = 16）

LinkedList:

linkedList 是一个双向链表，没有初始化大小，也没有扩容的机制，就是一直在前面或者后面新增就好。

ArrayList 与 Vector 的区别和适用场景

ArrayList 有三个构造方法：

```
public ArrayList(intinitialCapacity)// 构造一个具有指定初始容量的空列表。

public ArrayList()// 构造一个初始容量为 10 的空列表。
```

```
public ArrayList(Collection<? extends E> c)// 构造一个包含指定 collection 的元素的列表
```

Vector 有四个构造方法：

```
public Vector() // 使用指定的初始容量和等于零的容量增量构造一个空向量。

public Vector(int initialCapacity) // 构造一个空向量，使其内部数据数组的大小，其标准容量增量为零。

public Vector(Collection<? extends E> c)// 构造一个包含指定 collection 中的元素的向量

public Vector(int initialCapacity, int capacityIncrement)// 使用指定的初始容量和容量增量构造一个空的向量
```

ArrayList 和 Vector 都是用数组实现的，主要有这么四个区别：

1)Vector 是多线程安全的，线程安全就是说多线程访问代码，不会产生不确定的结果。而 ArrayList 不是，这可以从源码中看出，Vector 类中的方法很多有 synchronied 进行修饰，这样就导致了 Vector 在效率上无法与 ArrayLst 相比；

2)两个都是采用的线性连续空间存储元素，但是当空间充足的时候，两个类的增加方式是不同。

3)Vector 可以设置增长因子，而 ArrayList 不可以。

4)Vector 是一种老的动态数组，是线程同步的，效率很低，一般不赞成使用。

适用场景：

1.Vector 是线程同步的，所以它也是线程安全的，而 ArraList 是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用 ArrayList 效率比较高。

2.如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用 Vector 有一定的优势。

HashSet 与 TreeSet 的区别和适用场景

1.TreeSet 是二叉树（红黑树的树据结构）实现的，Treest 中的数据是自动排好序的，不允许放入 null 值。

2.HashSet 是哈希表实现的，HashSet 中的数据是无序的可以放入 null，但只能放入一个 null，两者中的值都不重复，就如数据库中唯一约束。

3.HashSet 要求放入的对象必须实现 HashCode()方法，放的对象，是以 hashcode 码作为标识的，而具有相同内容的 String 对象，hashcode 是一样，所以放入的内容不能重复但是同一个类的对象可以放入不同的实例。

适用场景分析:

HashSet 是基于 Hash 算法实现的，其性能通常都优于 TreeSet。为快速查找而设计的 Set，我们通常都应该使用 HashSet，在我们需要排序的功能时，我们才使用 TreeSet。

HashMap 与 TreeMap、HashTable 的区别及适用场景

HashMap 非线程安全

HashMap：基于哈希表(散列表)实现。使用 HashMap 要求的键类明确定义了 hashCode()和 equals()[可以重写 hasCode()和 equals()]，为了优化 HashMap 空间的使用，您可以调优初始容量和负载因子。其中散列表的冲突处理主分两种，一种是开放定址法，另一种是链表法。HashMap 实现中采用的是链表法。

TreeMap：非线程安全基于红黑树实现。TreeMap 没有调优选项，因为该树总处于平衡状态。

适用场景分析：

HashMap 和 HashTable:HashMap 去掉了 HashTable 的 contain 方法，但是加上了 containsValue()和 containsKey()方法。HashTable 是同步的，而 HashMap 是非同步的，效率上比 HashTable 要高。HashMap 允许空键值，而 HashTable 不允许。

HashMap：适用于 Map 中插入、删除和定位元素。

Treemap：适用于按自然顺序或自定义顺序遍历键(key)。 (ps:其实我们工作的过程中对集合的使用是很频繁的,稍注意并总结积累一下,在面试的时候应该会回答的很轻松)

2、set 集合从原理上如何保证不重复？

1）在往 set 中添加元素时，如果指定元素不存在，则添加成功。

2）具体来讲：当向 HashSet 中添加元素的时候，首先计算元素的 hashcode 值，然后用这个（元素的 hashcode）%（HashMap 集合的大小）+1 计算出这个元素的存储位置，如果这个位置为空，就将元素添加进去；如果不为空，则用 equals 方法比较元素是否相等，相等就不添加，否则找一个空位添加。

3、HashMap 和 HashTable 的主要区别是什么？，两者底层实现的数据结构是什么？

HashMap 和 HashTable 的区别：

二者都实现了 Map 接口，是将唯一的键映射到特定的值上，主要区别在于：

1)HashMap 没有排序，允许一个 null 键和多个 null 值,而 Hashtable 不允许；

2)HashMap 把 Hashtable 的 contains 方法去掉了，改成 containsvalue 和 containsKey, 因为 contains 方法容易让人引起误解；

3)Hashtable 继承自 Dictionary 类，HashMap 是 Java1.2 引进的 Map 接口的实现；

4)Hashtable 的方法是 Synchronized 的，而 HashMap 不是，在多个线程访问 Hashtable 时，不需要自己为它的方法实现同步，而 HashMap 就必须为之提供额外的同步。Hashtable 和 HashMap 采用的 hash/rehash 算法大致一样，所以性能不会有很大的差异。

HashMap 和 HashTable 的底层实现数据结构：

HashMap 和 Hashtable 的底层实现都是数组 + 链表结构实现的（jdk8 以前）

4、HashMap、ConcurrentHashMap、hash()相关原理解析？

HashMap 1.7 的原理：

HashMap 底层是基于 数组 + 链表 组成的，不过在 jdk1.7 和 1.8 中具体实现稍有不同。

负载因子：

- 给定的默认容量为 16，负载因子为 0.75。Map 在使用过程中不断的往里面存放数据，当数量达到了 16 * 0.75 = 12 就需要将当前 16 的容量进行扩容，而扩容这个过程涉及到 rehash、复制数据等操作，所以非常消耗性能。
- 因此通常建议能提前预估 HashMap 的大小最好，尽量的减少扩容带来的性能损耗。

其实真正存放数据的是 Entry<K,V>[] table，Entry 是 HashMap 中的一个静态内部类，它有 key、value、next、hash（key 的 hashcode）成员变量。

put 方法：

- 判断当前数组是否需要初始化。
- 如果 key 为空，则 put 一个空值进去。
- 根据 key 计算出 hashcode。
- 根据计算出的 hashcode 定位出所在桶。
- 如果桶是一个链表则需要遍历判断里面的 hashcode、key 是否和传入 key 相等，如果相等则进行覆盖，并返回原来的值。
- 如果桶是空的，说明当前位置没有数据存入，新增一个 Entry 对象写入当前位置。（当调用 addEntry 写入 Entry 时需要判断是否需要扩容。如果需要就进行两倍扩充，并将当前的 key 重新 hash 并定位。而在 createEntry 中会将当前位置的桶传入到新建的桶中，如果当前桶有值就会在位置形成链表。）

get 方法：

- 首先也是根据 key 计算出 hashcode，然后定位到具体的桶中。
- 判断该位置是否为链表。
- 不是链表就根据 key、key 的 hashcode 是否相等来返回值。
- 为链表则需要遍历直到 key 及 hashcode 相等时候就返回值。
- 啥都没取到就直接返回 null 。

HashMap 1.8 的原理：

当 Hash 冲突严重时，在桶上形成的链表会变的越来越长，这样在查询时的效率就会越来越低；时间复杂度为 O(N)，因此 1.8 中重点优化了这个查询效率。

TREEIFY_THRESHOLD 用于判断是否需要将链表转换为红黑树的阈值。

HashEntry 修改为 Node。

put 方法：

- 判断当前桶是否为空，空的就需要初始化（在 resize 方法 中会判断是否进行初始化）。
- 根据当前 key 的 hashcode 定位到具体的桶中并判断是否为空，为空表明没有 Hash 冲突就直接在当前位置创建一个新桶即可。
- 如果当前桶有值（ Hash 冲突），那么就要比较当前桶中的 key、key 的 hashcode 与写入的 key 是否相等，相等就赋值给 e,在第 8 步的时候会统一进行赋值及返回。
- 如果当前桶为红黑树，那就要按照红黑树的方式写入数据。
- 如果是个链表，就需要将当前的 key、value 封装成一个新节点写入到当前桶的后面（形成链表）。
- 接着判断当前链表的大小是否大于预设的阈值，大于时就要转换为红黑树。
- 如果在遍历过程中找到 key 相同时直接退出遍历。
- 如果 e != null 就相当于存在相同的 key,那就需要将值覆盖。
- 最后判断是否需要进行扩容。

get 方法：

- 首先将 key hash 之后取得所定位的桶。
- 如果桶为空则直接返回 null 。
- 否则判断桶的第一个位置(有可能是链表、红黑树)的 key 是否为查询的 key，是就直接返回 value。
- 如果第一个不匹配，则判断它的下一个是红黑树还是链表。
- 红黑树就按照树的查找方式返回值。

- 不然就按照链表的方式遍历匹配返回值。

修改为红黑树之后查询效率直接提高到了 O(logn)。但是 HashMap 原有的问题也都存在，比如在并发场景下使用时容易出现死循环：

- 在 HashMap 扩容的时候会调用 resize() 方法，就是这里的并发操作容易在一个桶上形成环形链表；这样当获取一个不存在的 key 时，计算出的 index 正好是环形链表的下标就会出现死循环：在 1.7 中 hash 冲突采用的头插法形成的链表，在并发条件下会形成循环链表，一旦有查询落到了这个链表上，当获取不到值时就会死循环。

ConcurrentHashMap 1.7 原理：

ConcurrentHashMap 采用了分段锁技术，其中 Segment 继承于 ReentrantLock。不会像 HashTable 那样不管是 put 还是 get 操作都需要做同步处理，理论上 ConcurrentHashMap 支持 CurrencyLevel (Segment 数组数量)的线程并发。每当一个线程占用锁访问一个 Segment 时，不会影响到其他的 Segment。

put 方法:

首先是通过 key 定位到 Segment，之后在对应的 Segment 中进行具体的 put。

虽然 HashEntry 中的 value 是用 volatile 关键词修饰的，但是并不能保证并发的原子性，所以 put 操作时仍然需要加锁处理。

首先第一步的时候会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用 scanAndLockForPut() 自旋获取锁:

尝试自旋获取锁。 如果重试的次数达到了 MAX_SCAN_RETRIES 则改为阻塞锁获取，保证能获取成功。

将当前 Segment 中的 table 通过 key 的 hashcode 定位到 HashEntry。

遍历该 HashEntry，如果不为空则判断传入的 key 和当前遍历的 key 是否相等，相等则覆盖旧的 value。

为空则需要新建一个 HashEntry 并加入到 Segment 中，同时会先判断是否需要扩容。

最后会使用 unlock()解除当前 Segment 的锁。

get 方法：

- 只需要将 Key 通过 Hash 之后定位到具体的 Segment ，再通过一次 Hash 定位到具体的元素上。
- 由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。
- ConcurrentHashMap 的 get 方法是非常高效的，因为整个过程都不需要加锁。

ConcurrentHashMap 1.8 原理：

1.7 已经解决了并发问题，并且能支持 N 个 Segment 这么多次数的并发，但依然存在 HashMap 在 1.7 版本中的问题：那就是查询遍历链表效率太低。和 1.8 HashMap 结构类似：其中抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。

CAS：

如果 obj 内的 value 和 expect 相等，就证明没有其他线程改变过这个变量，那么就更新它为 update，如果这一步的 CAS 没有成功，那就采用自旋的方式继续进行 CAS 操作。

问题：

- 目前在 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。这个类的 compareAndSet 方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。
- 如果 CAS 不成功，则会原地自旋，如果长时间自旋会给 CPU 带来非常大的执行开销。

put 方法：

- 根据 key 计算出 hashcode 。
- 判断是否需要进行初始化。
- 如果当前 key 定位出的 Node 为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功。
- 如果当前位置的 hashcode == MOVED == -1,则需要进行扩容。
- 如果都不满足，则利用 synchronized 锁写入数据。
- 最后，如果数量大于 TREEIFY_THRESHOLD 则要转换为红黑树。

get 方法：

- 根据计算出来的 hashcode 寻址，如果就在桶上那么直接返回值。

- 如果是红黑树那就按照树的方式获取值。
- 就不满足那就按照链表的方式遍历获取值。

1.8 在 1.7 的数据结构上做了大的改动，采用红黑树之后可以保证查询效率（O(logn)），甚至取消了 ReentrantLock 改为了 synchronized，这样可以看出在新版的 JDK 中对 synchronized 优化是很到位的。

HashMap、ConcurrentHashMap 1.7/1.8 实现原理

hash()算法全解析

HashMap 何时扩容：

当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即大于当前数组的长度乘以加载因子的值的时候，就要自动扩容。

扩容的算法是什么：

扩容(resize)就是重新计算容量，向 HashMap 对象里不停的添加元素，而 HashMap 对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然 Java 里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组。

Hashmap 如何解决散列碰撞（必问）？

Java 中 HashMap 是利用“拉链法”处理 HashCode 的碰撞问题。在调用 HashMap 的 put 方法或 get 方法时，都会首先调用 hashcode 方法，去查找相关的 key，当有冲突时，再调用 equals 方法。hashMap 基于 hasing 原理，我们通过 put 和 get 方法存取对象。当我们将键值对传递给 put 方法时，他调用键对象的 hashCode()方法来计算 hashCode，然后找到 bucket（哈希桶）位置来存储对象。

当获取对象时，通过键对象的 equals()方法找到正确的键值对，然后返回值对象。HashMap 使用链表来解决碰撞问题，当碰撞发生了，对象将会存储在链表的下一个节点中。hashMap 在每个链表节点存储键值对对象。当两个不同的键却有相同的 hashCode 时，他们会存储在同一个 bucket 位置的链表中。键对象的 equals()来找到键值对。

Hashmap 底层为什么是线程不安全的？

- 并发场景下使用时容易出现死循环，在 HashMap 扩容的时候会调用 resize() 方法，就是这里的并发操作容易在一个桶上形成环形链表；这样当获取一个不存在的 key 时，计算出的 index 正好是环形链表的下标就会出现死循环；
- 在 1.7 中 hash 冲突采用的头插法形成的链表，在并发条件下会形成循环链表，一旦有查询落到了这个链表上，当获取不到值时就会死循环。

5、ArrayMap 跟 SparseArray 在 HashMap 上面的改进？

HashMap 要存储完这些数据将要不断的扩容，而且在此过程中也需要不断的做 hash 运算，这将对我们的内存空间造成很大消耗和浪费。

SparseArray:

SparseArray 比 HashMap 更省内存，在某些条件下性能更好，主要是因为它避免了对 key 的自动装箱（int 转为 Integer 类型），它内部则是通过两个数组来进行数据存储的，一个存储 key，另外一个存储 value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间，我们从源码中可以看到 key 和 value 分别是用数组表示：

```
private int[] mKeys;
```

```
private Object[] mValues;
```

同时，SparseArray 在存储和读取数据时候，使用的是二分查找法。也就是在 put 添加数据的时候，会使用二分查找法和之前的 key 比较当前我们添加的元素的 key 的大小，然后按照从小到大的顺序排列好，所以，SparseArray 存储的元素都是按元素的 key 值从小到大排列好的。 而在获取数据的时候，也是使用二分查找法判断元素的位置，所以，在获取数据的时候非常快，比 HashMap 快的多。

ArrayMap:

ArrayMap 利用两个数组，mHashes 用来保存每一个 key 的 hash 值，mArrray 大小为 mHashes 的 2 倍，依次保存 key 和 value。

```
mHashes[index] = hash;

mArray[index<<1] = key;

mArray[(index<<1)+1] = value;
```

当插入时，根据 key 的 hashcode()方法得到 hash 值，计算出在 mArrays 的 index 位置，然后利用二分查找找到对应的位置进行插入，当出现哈希冲突时，会在 index 的相邻位置插入。

假设数据量都在千级以内的情况下：

1、如果 key 的类型已经确定为 int 类型，那么使用 SparseArray，因为它避免了自动装箱的过程，如果 key 为 long 类型，它还提供了一个 LongSparseArray 来确保 key 为 long 类型时的使用

2、如果 key 类型为其它的类型，则使用 ArrayMap。

三、反射 （★ ★ ★ ）

1、说说你对 Java 反射的理解？

答：Java 中的反射首先是能够获取到 Java 中要反射类的字节码， 获取字节码有三种方法：

1.Class.forName(className)

2.类名.class

3.this.getClass()。

然后将字节码中的方法，变量，构造函数等映射成相应的 Method、Filed、Constructor 等类，这些类提供了丰富的方法可以被我们所使用。

深入解析 Java 反射（1） - 基础

Java 基础之—反射（非常重要）

四、泛型 （★★）

1、简单介绍一下 java 中的泛型，泛型擦除以及相关的概念，解析与分派？

泛型是 Java SE1.5 的新特性，泛型的本质是参数化类型，也就是说所操的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。 Java 语言引入泛型的好处是安全简单。

在 Java SE 1.5 之前，没有泛型的情况的下，通过对类型 Object 的引用来实现参数的“任意化”，“任意化”带来的缺点是要做显式的强制类型转换，而这种转换是要求开发者实际参数类型可以预知的情况下进行的。对于强制类型换错误的情况，编译器可能不提示错误，在运行的时候出现异常，这是一个安全隐患。

泛型的好处是在编译的时候检查类型安全，并且所有的转换都是自动和隐式的，提高代码的重用率。

1、泛型的类型参数只能是类类型（包括自定义类），不是简单类型。

2、同一种泛型可以对应多个版本（因为参数类型是不确的），不同版本的泛型类实例是不兼容的。

3、泛型的类型参数可以有多个。

4、泛型的参数类型可以使用 extends 语句，例如。习惯上称为“有界类型”。

5、泛型的参数类型还可以是通配符类型。例如 Class<?> classType =

Class.forName("java.lang.String");

泛型擦除以及相关的概念

泛型信息只存在代码编译阶段，在进入 JVM 之前，与泛型关的信息都会被擦除掉。

在类型擦除的时候，如果泛型类里的类型参数没有指定上限，则会被转成 Object 类型，如果指定了上限，则会被传转换成对应的类型上限。

Java 中的泛型基本上都是在编译器这个层次来实现的。生成的 Java 字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会在编译器在编译的时候擦除掉。这个过程就称为类型擦除。

类型擦除引起的问题及解决方法：

1、先检查，在编译，以及检查编译的对象和引用传递的题

2、自动类型转换

3、类型擦除与多态的冲突和解决方法

4、泛型类型变量不能是基本数据类型

5、运行时类型查询

6、异常中使用泛型的问题

7、数组（这个不属于类型擦除引起的问题）

9、类型擦除后的冲突

10、泛型在静态方法和静态类中的问题

五、注解 （★★）

1、说说你对 Java 注解的理解？

注解相当于一种标记，在程序中加了注解就等于为程序打上了某种标记。程序可以利用 ava 的反射机制来了解你的类及各种元素上有无何种标记，针对不同的标记，就去做相应的事件。标记可以加在包，类，字段，方法，方法的参数以及局部变量上。

六、其它 （★★）

1、Java 的 char 是两个字节，是怎么存 Utf-8 的字符的？

是否熟悉 Java char 和字符串（初级）

- char 是 2 个字节，utf-8 是 1~3 个字节。
- 字符集（字符集不是编码）：ASCII 码与 Unicode 码。
- 字符 -> 0xd83dde00(码点）。

是否了解字符的映射和存储细节（中级）

人类认知：字符 => 字符集：0x4e2d(char) => 计算机存储(byte)：01001110:4e、00101101:2d

编码：UTF-16

“中”.getBytes("utf-6"); -> fe ff 4e 2d：4 个字节，其中前面的 fe ff 只是字节序标志。

是否能触类旁通，横向对比其他语言（高级）

Python2 的字符串：

- byteString = "中"
- unicodeString = u"中"

令人迷惑的字符串长度

```
emoij = u"表情"

print(len(emoji)
```

Java 与 python 3.2 及以下：2 字节 python >= 3.3：1 字节

注意：Java 9 对 latin 字符的存储空间做了优化，但字符串长度还是!= 字符数。

总结

- Java char 不存 UTF-8 的字节，而是 UTF-16。
- Unicode 通用字符集占两个字节，例如“中”。
- Unicode 扩展字符集需要用一对 char 来表示，例如“表情”。
- Unicode 是字符集，不是编码，作用类似于 ASCII 码。
- Java String 的 length 不是字符数。

2、Java String 可以有多长？

是否对字符串编解码有深入了解（中级）

分配到栈：

```
String longString = "aaa...aaa";
```

分配到堆：

```
byte[] bytes = loadFromFile(new File("superLongText.txt");

String superLongString = new String(bytes);
```

是否对字符串在内存当中的存储形式有深入了解（高级）

是否对 Java 虚拟机字节码有足够的了解（高级）

源文件：*.java

```
String longString = "aaa...aaa";

字节数 <= 65535
```

字节码：*.class

```
CONSTANT_Utf8_info {

    u1 tag;

    u2 length;

    (0~65535) u1 bytes[length];

    最多 65535 个字节

}
```

javac 的编译器有问题，< 65535 应该改为< = 65535。

Java String 栈分配

- 受字节码限制，字符串最终的 MUTF-8 字节数不超过 65535。
- Latin 字符，受 Javac 代码限制，最多 65534 个。
- 非 Latin 字符最终对应字节个数差异较大，最多字节个数是 65535。

- 如果运行时方法区设置较小，也会受到方法区大小的限制。

是否对 java 虚拟机指令有一定的认识（高级）

new String(bytes)内部是采用了一个字符数组，其对应的虚拟机指令是 newarray [int] ，数组理论最大个数为 Integer.MAX_VALUE，有些虚拟机需要一些头部信息，所以 MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8。

Java String 堆分配

- 受虚拟机指令限制，字符数理论上限为 Integer.MAX_VALUE。
- 受虚拟机实现限制，实际上限可能会小于 Integer.MAX_VALUE。
- 如果堆内存较小，也会受到堆内存的限制。

总结

Java String 字面量形式

- 字节码中 CONSTANT_Utf8_info 的限制
- Javac 源码逻辑的限制
- 方法区大小的限制

Java String 运行时创建在堆上的形式

- Java 虚拟机指令 newarray 的限制
- Java 虚拟机堆内存大小的限制

3、Java 的匿名内部类有哪些限制？

考察匿名内部类的概念和用法（初级）

- 匿名内部类的名字：没有人类认知意义上的名字

- 只能继承一个父类或实现一个接口
- 包名.OuterClass$1，表示定位的第一个匿名内部类。外部类加$N，N 是匿名内部类的顺序。

考察语言规范以及语言的横向对比等（中级）

匿名内部类的继承结构：Java 中的匿名内部类不可以继承，只有内部类才可以有实现继承、实现接口的特性。而 Kotlin 是的匿名内部类是支持继承的，如

```
val runnableFoo = object: Foo(),Runnable {
       override fun run() {


       }
}
```

作为考察内存泄漏的切入点（高级）

匿名内部类的构造方法（深入源码字节码探索语言本质的能力）：

- 匿名内部类会默认持有外部类的引用，可能会导致内存泄漏。
- 由编译器生成的。

其参数列表包括

- 外部对象（定义在非静态域内）
- 父类的外部对象（父类非静态）
- 父类的构造方法参数（父类有构造方法且参数列表不为空）
- 外部捕获的变量（方法体内有引用外部 final 变量）

Lambda 转换(SAM 类型，仅支持单一接口类型）：

如果 CallBack 是一个 interface，不是抽象类，则可以转换为 Lambda 表达式。

```
CallBack callBack = () -> {
       ...
```

```
};
```

总结

- 没有人类认知意义上的名字。
- 只能继承一个父类或实现一个接口。
- 父类是非静态的类型，则需父类外部实例来初始化。
- 如果定义在非静态作用域内，会引用外部类实例。
- 只能捕获外部作用域内的 final 变量。
- 创建时只有单一方法的接口可以用 Lambda 转换。

技巧点拨

关注语言版本的变化：

- 体现对技术的热情
- 体现好学的品质
- 显得专业

4、Java 中对异常是如何进行分类的？

异常整体分类：

Java 异常结构中定义有 Throwable 类。 Exception 和 Error 为其子类。

Error 是程序无法处理的错误，比如 OutOfMemoryError、StackOverflowError。

这些异常发生时， Java 虚拟机（JVM）一般会选择线程终止。

Exception 是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常，程序中应当尽可能去处理这些异常。

运行时异常都是 RuntimeException 类及其子类异常，如 NullPointerException、IndexOutOfBoundsException 等， 这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的， 程序应该从逻辑角度尽可能避免这类异常的发生。

异常处理的两个基本原则:

1、尽量不要捕获类似 Exception 这样的通用异常，而是应该捕获特定异常。

2、不要生吞异常。

NoClassDefFoundError 和 ClassNotFoundException 有什么区别？

ClassNotFoundException 的产生原因主要是： Java 支持使用反射方式在运行时动态加载类，例如使用 Class.forName 方法来动态地加载类时，可以将类名作为参数传递给上述方法从而将指定类加载到 JVM 内存中，如果这个类在类路径中没有被找到，那么此时就会在运行时抛出 ClassNotFoundException 异常。 解决该问题需要确保所需的类连同它依赖的包存在于类路径中，常见问题在于类名书写错误。 另外还有一个导致 ClassNotFoundException 的原因就是：当一个类已经某个类加载器加载到内存中了，此时另一个类加载器又尝试着动态地从同一个包中加载这个类。通过控制动态类加载过程，可以避免上述情况发生。

NoClassDefFoundError 产生的原因在于： 如果 JVM 或者 ClassLoader 实例尝试加载（可以通过正常的方法调用，也可能是使用 new 来创建新的对象）类的时候却找不到类的定义。要查找的类在编译的时候是存在的，运行的时候却找不到了。这个时候就会导致 NoClassDefFoundError. 造成该问题的原因可能是打包过程漏掉了部分类，或者 jar 包出现损坏或者篡改。解决这个问题的办法是查找那些在开发期间存在于类路径下但在运行期间却不在类路径下的类。

5、String 为什么要设计成不可变的？

String 是不可变的（修改 String 时，不会在原有的内存地址修改，而是重新指向一个新对象），String 用 final 修饰，不可继承，String 本质上是个 final 的 char[]数组，所以 char[]数组的内存地址不会被修改，而且 String 也没有对外暴露修改 char[]数组的方法。不可变性可以保证线程安全以及字符串串常量池的实现。

6、Java 里的幂等性了解吗？

幂等性原本是数学上的一个概念，即：f(x) = f(f(x))，对同一个系统，使用同样的条件，一次请求和重复的多次请求对系统资源的影响是一致的。

幂等性最为常见的应用就是电商的客户付款，试想一下如果你在付款的时候因为网络等各种问题失败了，然后去重复的付了一次，是一种多么糟糕的体验。幂等性就是为了解决这样的问题。

实现幂等性可以使用 Token 机制。

核心思想是为每一次操作生成一个唯一性的凭证，也就是 token。一个 token 在操作的每一个阶段只有一次执行权，一旦执行成功则保存执行结果。对重复的请求，返回同一个结果。

例如：电商平台上的订单 id 就是最适合的 token。当用户下单时，会经历多个环节，比如生成订单，减库存，减优惠券等等。每一个环节执行时都先检测一下该订单 id 是否已经执行过这一步骤，对未执行的请求，执行操作并缓存结果，而对已经执行过的 id，则直接返回之前的执行结果，不做任何操 作。这样可以在最大程度上避免操作的重复执行问题，缓存起来的执行结果也能用于事务的控制等。

7、为什么 Java 里的匿名内部类只能访问 final 修饰的外部变量？

匿名内部类用法：

```
public class TryUsingAnonymousClass {

   public void useMyInterface() {

       final Integer number = 123;

       System.out.println(number);



       MyInterface myInterface = new MyInterface() {

           @Override

           public void doSomething() {
```

```
                System.out.println(number);

            }

        };

        myInterface.doSomething();



        System.out.println(number);

    }

}
```

编译后的结果

```
class TryUsingAnonymousClass$1

        implements MyInterface {

    private final TryUsingAnonymousClass this$0;

    private final Integer paramInteger;



    TryUsingAnonymousClass$1(TryUsingAnonymousClass this$0, Integer
paramInteger) {

        this.this$0 = this$0;

        this.paramInteger = paramInteger;

    }



    public void doSomething() {

        System.out.println(this.paramInteger);

    }

}
```

因为匿名内部类最终会编译成一个单独的类，而被该类使用的变量会以构造函数参数的形式传递给该类，例如：Integer paramInteger，如果变量不定义成 final

的，paramInteger 在匿名内部类被可以被修改，进而造成和外部的 paramInteger 不一致的问题，为了避免这种不一致的情况，因次 Java 规定匿名内部类只能访问 final 修饰的外部变量。

8、讲一下 Java 的编码方式？

为什么需要编码

计算机存储信息的最小单元是一个字节即 8bit，所以能示的范围是 0~255，这个范围无法保存所有的字符，所以要一个新的数据结构 char 来表示这些字符，从 char 到 byte 需要编码。

常见的编码方式有以下几种：

ASCII：总共有 128 个，用一个字节的低 7 位表示，031 是控制字符如换行回车删除等；32126 是打印字符，可以通过键盘输入并且能够显示出来。

GBK：码范围是 8140~FEFE（去掉 XX7F）总共有 23940 个码位，它能表示 21003 个汉字，它的编码是和 GB2312 兼容的，也就是说用 GB2312 编码的汉字可以用 GBK 来解码，并且不会有乱码。

UTF-16：UTF-16 具体定义了 Unicode 字符在计算机中存取方法。UTF-16 用两个字节来表示 Unicode 转化格式，这个是定长的表示方法，不论什么字符都可以用两个字节表示，两个字节是 16 个 bit，所以叫 UTF-16。UTF-16 表示字符非常方便，每两个字节表示一个字符，这个在字符串操作时就大大简化了操作，这也是 Java 以 UTF-16 作为内存的字符存储格式的一个很重要的原因。

UTF-8：统一采用两个字节表示一个字符，虽然在表示上非常简单方便，但是也有其缺点，有很大一部分字符用一个字节就可以表示的现在要两个字节表示，存储空间放大了一倍，在现在的网络带宽还非常有限的今天，这样会增大网络传输

的流量，而且也没必要。而 UTF-8 采用了一种变长技术，每个编码区域有不同的字码长度。不同类型的字符可以是由 1~6 个字节组成。

Java 中需要编码的地方一般都在字符到字节的转换上，这个一般包括磁盘 IO 和网络 IO。

Reader 类是 Java 的 I/O 中读字符的父类，而 InputStream 类是读字节的父类，InputStreamReader 类就是关联字节到字符的桥梁，它负责在 I/O 过程中处理读取字节到字符的转换，而具体字节到字符解码实现由 StreamDecoder 去实现，在 StreamDecoder 解码过程中必须由用户指定 Charset 编码格式。

9、String，StringBuffer，StringBuilder 有哪些不同？

三者在执行速度方面的比较：StringBuilder > StringBuffer > String

String 每次变化一个值就会开辟一个新的内存空间

StringBuilder：线程非安全的

StringBuffer：线程安全的

对于三者使用的总结：

1.如果要操作少量的数据用 String。

2.单线程操作字符串缓冲区下操作大量数据用 StringBuilder。

3.多线程操作字符串缓冲区下操作大量数据用 StringBuffer。

String 是 Java 语言非常基础和重要的类，提供了构造和管理字符串的各种基本逻辑。它是典型的 Immutable 类，被声明成为 final class，所有属性也都是 final 的。也由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的 String 对象。由于字符串操作的普遍性，所以相关操作的效率往往对应用性能有明显影响。

StringBuffer 是为解决上面提到拼接产生太多中间对象的问题而提供的一个类，我们可以用 append 或者 add 方法，把字符串添加到已有序列的末尾或者指定位置。StringBuffer 本质是一个线程安全的可修改字符序列，它保证了线程安全，也随之带来了额外的性能开销，所以除非有线程安全的需要，不然还是推荐使用它的后继者，也就是 StringBuilder。

StringBuilder 是 Java 1.5 中新增的，在能力上和 StringBuffer 没有本质区别，但是它去掉了线程安全的部分，有效减小了开销，是绝大部分情况下进行字符串拼接的首选。

10、什么是内部类？内部类的作用。

内部类可以有多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。

在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。

创建内部类对象并不依赖于外围类对象的创建。

内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

内部类提供了更好的封装，除了该外围类，其他类都不能访问。。

11、抽象类和接口区别？

共同点

- 是上层的抽象层。
- 都不能被实例化。

- 都能包含抽象的方法，这些抽象的方法用于描述类具备的功能，但是不提供具体的实现。

区别：

- 1、在抽象类中可以写非抽象的方法，从而避免在子类中重复书写他们，这样可以提高代码的复用性，这是抽象类的优势，接口中只能有抽象的方法。
- 2、多继承：一个类只能继承一个直接父类，这个父类可以是具体的类也可是抽象类，但是一个类可以实现多个接口。
- 3、抽象类可以有默认的方法实现，接口根本不存在方法的实现。
- 4、子类使用 extends 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明方法的实现。子类使用关键字 implements 来实现接口。它需要提供接口中所有声明方法的实现。
- 5、构造器：抽象类可以有构造器，接口不能有构造器。
- 6、和普通 Java 类的区别：除了你不能实例化抽象类之外，抽象类和普通 Java 类没有任何区别，接口是完全不同的类型。
- 7、访问修饰符:抽象方法可以有 public、protected 和 default 修饰符，接口方法默认修饰符是 public。你不可以使用其它修饰符。
- 8、main 方法:抽象方法可以有 main 方法并且我们可以运行它接口没有 main 方法，因此我们不能运行它。
- 9、速度:抽象类比接口速度要快，接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
- 10、添加新方法:如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。如果你往接口中添加方法，那么你必须改变实现该接口的类。

12、接口的意义？

规范、扩展、回调。

13、父类的静态方法能否被子类重写？

不能。子类继承父类后，用相同的静态方法和非静态方法，这时非静态方法覆盖父类中的方法（即方法重写），父类的该静态方法被隐藏（如果对象是父类则调用该隐藏的方法），另外子类可继承父类的静态与非静态方法，至于方法重载我觉得它其中一要素就是在同一类中，不能说父类中的什么方法与子类里的什么方法是方法重载的体现。

14、抽象类的意义？

为其子类提供一个公共的类型，封装子类中的重复内容，定义抽象方法，子类虽然有不同的实现 但是定义是一致的。

15、静态内部类、非静态内部类的理解？

静态内部类：只是为了降低包的深度，方便类的使用，静态内部类适用于包含在类当中，但又不依赖与外在的类，不用使用外在类的非静态属性和方法，只是为了方便管理类结构而定义。在创建静态内部类的时候，不需要外部类对象的引用。

非静态内部类：持有外部类的引用，可以自由使用外部类的所有变量和方法。

16、为什么复写 equals 方法的同时需要复写 hashcode 方法，前者相同后者是否相同，反过来呢？为什么？

要考虑到类似 HashMap、HashTable、HashSet 的这种散列的数据类型的运用，

当我们重写 equals 时，是为了用自身的方式去判断两个自定义对象是否相等，

然而如果此时刚好需要我们用自定义的对象去充当 hashmap 的键值使用时，就

会出现我们认为的同一对象，却因为 hash 值不同而导致 hashmap 中存了两个对

象，从而才需要进行 hashcode 方法的覆盖。

17、equals 和 hashcode 的关系？

hashcode 和 equals 的约定关系如下：

1、如果两个对象相等，那么他们一定有相同的哈希值（hashcode）。

2、如果两个对象的哈希值相等，那么这两个对象有可能相等也有可能不相等。（需要再通过 equals 来判断）

18、java 为什么跨平台？

因为 Java 程序编译之后的代码不是能被硬件系统直接运行的代码，而是一种“中间码”——字节码。然后不同的硬件平台上安装有不同的 Java 虚拟机(JVM)，由 JVM 来把字节码再“翻译”成所对应的硬件平台能够执行的代码。因此对于 Java 编程者来说，不需要考虑硬件平台是什么。所以 Java 可以跨平台。

19、浮点数的精准计算

BigDecimal 类进行商业计算，Float 和 Double 只能用来做科学计算或者是工程计算。

20、final，finally，finalize 的区别？

final 可以用来修饰类、方法、变量，分别有不同的意义，final 修饰的 class 代表不可以继承扩展，final 的变量是不可以修改的，而 final 的方法也是不可以重写的（override）。

finally 则是 Java 保证重点代码一定要被执行的一种机制。我们可以使用 try-finally 或者 try-catch-finally 来进行类似关闭 JDBC 连接、保证 unlock 锁等动作。

finalize 是基础类 java.lang.Object 的一个方法，它的设计目的是保证对象在被垃圾收集前完成特定资源的回收。finalize 机制现在已经不推荐使用，并且在 JDK 9 开始被标记为 deprecated。Java 平台目前在逐步使用 java.lang.ref.Cleaner 来替换掉原有的 finalize 实现。Cleaner 的实现利用了幻

象引用（PhantomReference），这是一种常见的所谓 post-mortem 清理机制。利用幻象引用和引用队列，我们可以保证对象被彻底销毁前做一些类似资源回收的工作，比如关闭文件描述符（操作系统有限的资源），它比 finalize 更加轻量、更加可靠。

21、静态内部类的设计意图

静态内部类与非静态内部类之间存在一个最大的区别：非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。

没有这个引用就意味着：

它的创建是不需要依赖于外围类的。 它不能使用任何外围类的非 static 成员变量和方法。

22、Java 中对象的生命周期

在 Java 中，对象的生命周期包括以下几个阶段：

1.创建阶段(Created)

JVM 加载类的 class 文件 此时所有的 static 变量和 static 代码块将被执行 加载完成后，对局部变量进行赋值（先父后子的顺序） 再执行 new 方法 调用构造函数 一旦对象被创建，并被分派给某些变量赋值，这个对象的状态就切换到了应用阶段。

2.应用阶段(In Use)

对象至少被一个强引用持有着。

3.不可见阶段(Invisible)

当一个对象处于不可见阶段时，说明程序本身不再持有该对象的任何强引用，虽然该这些引用仍然是存在着的。 简单说就是程序的执行已经超出了该对象的作用域了。

4.不可达阶段(Unreachable)

对象处于不可达阶段是指该对象不再被任何强引用所持有。 与“不可见阶段”相比，“不可见阶段”是指程序不再持有该对象的任何强引用，这种情况下，该对象仍可能被 JVM 等系统下的某些已装载的静态变量或线程或 JNI 等强引用持有着，这些特殊的强引用被称为”GC root”。存在着这些 GC root 会导致对象的内存泄露情况，无法被回收。

5.收集阶段(Collected)

当垃圾回收器发现该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备时，则对象进入了“收集阶段”。如果该对象已经重写了 finalize()方法，则会去执行该方法的终端操作。

6.终结阶段(Finalized)

当对象执行完 finalize()方法后仍然处于不可达状态时，则该对象进入终结阶段。在该阶段是等待垃圾回收器对该对象空间进行回收。

7.对象空间重分配阶段(De-allocated)

垃圾回收器对该对象的所占用的内存空间进行回收或者再分配了，则该对象彻底消失了，称之为“对象空间重新分配阶段。

23、静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？

结论：java 中静态属性和静态方法可以被继承，但是不可以被重写而是被隐藏。

原因：

1). 静态方法和属性是属于类的，调用的时候直接通过类名.方法名完成，不需要继承机制即可以调用。如果子类里面定义了静态方法和属性，那么这时候父类的

静态方法或属性称之为"隐藏"。如果你想要调用父类的静态方法和属性，直接通过父类名.方法或变量名完成，至于是否继承一说，子类是有继承静态方法和属性，但是跟实例方法和属性不太一样，存在"隐藏"的这种情况。

2). 多态之所以能够实现依赖于继承、接口和重写、重载（继承和重写最为关键）。有了继承和重写就可以实现父类的引用指向不同子类的对象。重写的功能是："重写"后子类的优先级要高于父类的优先级，但是"隐藏"是没有这个优先级之分的。

3). 静态属性、静态方法和非静态的属性都可以被继承和隐藏而不能被重写，因此不能实现多态，不能实现父类的引用可以指向不同子类的对象。非静态方法可以被继承和重写，因此可以实现多态。

24、object 类的 equal 和 hashcode 方法重写，为什么？

在 Java API 文档中关于 hashCode 方法有以下几点规定（原文来自 java 深入解析一书）：

1、在 java 应用程序执行期间，如果在 equals 方法比较中所用的信息没有被修改，那么在同一个对象上多次调用 hashCode 方法时必须一致地返回相同的整数。如果多次执行同一个应用时，不要求该整数必须相同。

2、如果两个对象通过调用 equals 方法是相等的，那么这两个对象调用 hashCode 方法必须返回相同的整数。

3、如果两个对象通过调用 equals 方法是不相等的，不要求这两个对象调用 hashCode 方法必须返回不同的整数。但是程序员应该意识到对不同的对象产生不同的 hash 值可以提供哈希表的性能。

25、java 中==和 equals 和 hashCode 的区别？

默认情况下也就是从超类 Object 继承而来的 equals 方法与'=='是完全等价的，比较的都是对象的内存地址，但我们可以重写 equals 方法，使其按照我们的需求的方式进行比较，如 String 类重写了 equals 方法，使其比较的是字符的序列，而不再是内存地址。在 java 的集合中，判断两个对象是否相等的规则是：

```
1.判断两个对象的 hashCode 是否相等。

2.判断两个对象用 equals 运算是否相等。
```

26、Java 的四种引用及使用场景？

- 强引用（FinalReference）：在内存不足时不会被回收。平常用的最多的对象，如新创建的对象。
- 软引用（SoftReference）：在内存不足时会被回收。用于实现内存敏感的高速缓存。
- 弱引用（WeakReferenc）：只要 GC 回收器发现了它，就会将之回收。用于 Map 数据结构中，引用占用内存空间较大的对象。
- 虚引用（PhantomReference）：在回收之前，会被放入 ReferenceQueue，JVM 不会自动将该 referent 字段值设置成 null。其它引用被 JVM 回收之后才会被放入 ReferenceQueue 中。用于实现一个对象被回收之前做一些清理工作。

27、类的加载过程，Person person = new Person();为例进行说明。

1).因为 new 用到了 Person.class，所以会先找到 Person.class 文件，并加载到内存中;

2).执行该类中的 static 代码块，如果有的话，给 Person.class 类进行初始化;

3).在堆内存中开辟空间分配内存地址;

4).在堆内存中建立对象的特有属性，并进行默认初始化;

5).对属性进行显示初始化;

6).对对象进行构造代码块初始化;

7).对对象进行与之对应的构造函数进行初始化;

8).将内存地址付给栈内存中的 p 变量。

28、JAVA 常量池

Interger 中的 128(-128~127)

a.当数值范围为-128~127 时：如果两个 new 出来的 Integer 对象，即使值相同，通过"=="比较结果为 false，但两个对直接赋值，则通过"=="比较结果为"true，这一点与 String 非常相似。

b.当数值不在-128~127 时，无论通过哪种方式，即使两对象的值相等，通过"=="比较，其结果为 false；

c.当一个 Integer 对象直接与一个 int 基本数据类型通过"=="比较，其结果与第一点相同；

d.Integer 对象的 hash 值为数值本身；

为什么是-128-127?

在 Integer 类中有一个静态内部类 IntegerCache，在 IntegrCache 类中有一个 Integer 数组，用以缓存当前数值范围为-128~127 时的 Integer 对象。

29、在重写 equals 方法时，需要遵循哪些约定，具体介绍一下？

重写 equals 方法时需要遵循通用约定：自反性、对称性、传递性、一致性、非空性

1）自反性

对于任何非 null 的引用值 x,x.equals(x)必须返回 true。---这一点基本上不会有啥问题

2）对称性

对于任何非 null 的引用值 x 和 y，当且仅当 x.equals(y)为 true 时，y.equals(x)也为 true。

3）传递性

对于任何非 null 的引用值 x、y、z。如果 x.equals(y)==true,y.equals(z)==true,那么 x.equals(z)==true。

4） 一致性

对于任何非 null 的引用值 x 和 y，只要 equals 的比较操作在对象所用的信息没有被修改，那么多次调用 x.equals(y)就会一致性地返回 true,或者一致性的返回 false。

5）非空性

所有比较的对象都不能为空。

30、深拷贝和浅拷贝的区别

31、Integer 类对 int 的优化

第二节 Java 并发面试题

一、线程池相关 （★ ★ ★ ）

1、什么是线程池，如何使用？为什么要使用线程池？

答：线程池就是事先将多个线程对象放到一个容器中，使用的时候就不用 new 线程而是直接去池中拿线程即可，节 省了开辟子线程的时间，提高了代码执行效率。

2、Java 中的线程池共有几种？

Java 有四种线程池：

第一种：newCachedThreadPool

不固定线程数量，且支持最大为 Integer.MAX_VALUE 的线程数量:

```
public static ExecutorService newCachedThreadPool() {
    // 这个线程池 corePoolSize 为 0，maximumPoolSize 为 Integer.MAX_VALUE
    // 意思也就是说来一个任务就创建一个 woker，回收时间是 60s
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

可缓存线程池：

1、线程数无限制。 2、有空闲线程则复用空闲线程，若无空闲线程则新建线程。

3、一定程序减少频繁创建/销毁线程，减少系统开销。

第二种：newFixedThreadPool

一个固定线程数量的线程池:

```
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {

    // corePoolSize 跟 maximumPoolSize 值一样，同时传入一个无界阻塞队列

    // 该线程池的线程会维持在指定线程数，不会进行回收

    return new ThreadPoolExecutor(nThreads, nThreads,

                                0L, TimeUnit.MILLISECONDS,

                                new LinkedBlockingQueue<Runnable>(),

                                threadFactory);

}
```

定长线程池：

1、可控制线程最大并发数（同时执行的线程数）。 2、超出的线程会在队列中等待。

第三种：newSingleThreadExecutor

可以理解为线程数量为 1 的 FixedThreadPool:

```
public static ExecutorService newSingleThreadExecutor() {

    // 线程池中只有一个线程进行任务执行，其他的都放入阻塞队列

    // 外面包装的 FinalizableDelegatedExecutorService 类实现了 finalize 方法，在
JVM 垃圾回收的时候会关闭线程池

    return new FinalizableDelegatedExecutorService

        (new ThreadPoolExecutor(1, 1,

                                0L, TimeUnit.MILLISECONDS,

                                new LinkedBlockingQueue<Runnable>()));

}
```

单线程化的线程池：

1、有且仅有一个工作线程执行任务。 2、所有任务按照指定顺序执行，即遵循队列的入队出队规则。

第四种：newScheduledThreadPool。

支持定时以指定周期循环执行任务:

```
public static ScheduledExecutorService newScheduledThreadPool(int
corePoolSize) {

    return new ScheduledThreadPoolExecutor(corePoolSize);

}
```

注意：前三种线程池是 ThreadPoolExecutor 不同配置的实例，最后一种是 ScheduledThreadPoolExecutor 的实例。

3、线程池原理？

从数据结构的角度来看，线程池主要使用了阻塞队列（BlockingQueue）和 HashSet 集合构成。 从任务提交的流程角度来看，对于使用线程池的外部来说，线程池的机制是这样的：

```
1、如果正在运行的线程数 < coreSize，马上创建核心线程执行该 task，不排队等待；

2、如果正在运行的线程数 >= coreSize，把该 task 放入阻塞队列；

3、如果队列已满 && 正在运行的线程数 < maximumPoolSize，创建新的非核心线程执行该
task；

4、如果队列已满 && 正在运行的线程数 >= maximumPoolSize，线程池调用 handler 的 reject
方法拒绝本次提交。
```

理解记忆：1-2-3-4 对应（核心线程->阻塞队列->非核心线程->handler 拒绝提交）。

线程池的线程复用：

这里就需要深入到源码 addWorker()：它是创建新线程的关键，也是线程复用的关键入口。最终会执行到 runWoker，它取任务有两个方式：

- firstTask：这是指定的第一个 runnable 可执行任务，它会在 Woker 这个工作线程中运行执行任务 run。并且置空表示这个任务已经被执行。
- getTask()：这首先是一个死循环过程，工作线程循环直到能够取出 Runnable 对象或超时返回，这里的取的目标就是任务队列 workQueue，对应刚才入队的操作，有入有出。

其实就是任务在并不只执行创建时指定的 firstTask 第一任务，还会从任务队列的中通过 getTask()方法自己主动去取任务执行，而且是有/无时间限定的阻塞等待，保证线程的存活。

信号量

semaphore 可用于进程间同步也可用于同一个进程间的线程同步。

可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号量。

4、线程池都有哪几种工作队列？

1、ArrayBlockingQueue

是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

2、LinkedBlockingQueue

一个基于链表结构的阻塞队列，此队列按 FIFO （先进先出） 排序元素，吞吐量通常要高于 ArrayBlockingQueue。静态工厂方法

Executors.newFixedThreadPool()和 Executors.newSingleThreadExecutor 使用了这个队列。

3、SynchronousQueue

一个不存储元素的阻塞队列。每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于 LinkedBlockingQueue，静态工厂方法 Executors.newCachedThreadPool 使用了这个队列。

4、PriorityBlockingQueue

一个具有优先级的无限阻塞队列。

5、怎么理解无界队列和有界队列？

有界队列

1.初始的 poolSize < corePoolSize，提交的 runnable 任务，会直接做为 new 一个 Thread 的参数，立马执行 。 2.当提交的任务数超过了 corePoolSize，会将当前的 runable 提交到一个 block queue 中。3.有界队列满了之后，如果 poolSize < maximumPoolsize 时，会尝试 new 一个 Thread 的进行救急处理，立马执行对应的 runnable 任务。 4.如果 3 中也无法处理了，就会走到第四步执行 reject 操作。

无界队列

与有界队列相比，除非系统资源耗尽，否则无界的任务队列不存在任务入队失败的情况。当有新的任务到来，系统的线程数小于 corePoolSize 时，则新建线程执行任务。当达到 corePoolSize 后，就不会继续增加，若后续仍有新的任务加入，而没有空闲的线程资源，则任务直接进入队列等待。若任务创建和处理的速度差异很大，无界队列会保持快速增长，直到耗尽系统内存。 当线程池的任务

缓存队列已满并且线程池中的线程数目达到 maximumPoolSize，如果还有任务到来就会采取任务拒绝策略。

6、多线程中的安全队列一般通过什么实现？

Java 提供的线程安全的 Queue 可以分为阻塞队列和非阻塞队列，其中阻塞队列的典型例子是 BlockingQueue，非阻塞队列的典型例子是 ConcurrentLinkedQueue.

对于 BlockingQueue，想要实现阻塞功能，需要调用 put(e) take() 方法。而 ConcurrentLinkedQueue 是基于链接节点的、无界的、线程安全的非阻塞队列。

二、Synchronized、volatile、Lock(ReentrantLock)相关 （★★★）

1、synchronized 的原理？

synchronized 代码块是由一对儿 monitorenter/monitorexit 指令实现的，Monitor 对象是同步的基本实现，而 synchronized 同步方法使用了 ACC_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

在 Java 6 之前，Monitor 的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作。

现代的（Oracle）JDK 中，JVM 对此进行了大刀阔斧地改进，提供了三种不同的 Monitor 实现，也就是常说的三种不同的锁：偏斜锁（Biased Locking）、轻量级锁和重量级锁，大大改进了其性能。

所谓锁的升级、降级，就是 JVM 优化 synchronized 运行的机制，当 JVM 检测到不同的竞争状况时，会自动切换到适合的锁实现，这种切换就是锁的升级、降级。

当没有竞争出现时，默认会使用偏斜锁。JVM 会利用 CAS 操作，在对象头上的 Mark Word 部分设置线程 ID，以表示这个对象偏向于当前线程，所以并不涉及真正的互斥锁。这样做的假设是基于在很多应用场景中，大部分对象生命周期中最多会被一个线程锁定，使用偏斜锁可以降低无竞争开销。

如果有另外的线程试图锁定某个已经被偏斜过的对象，JVM 就需要撤销（revoke）偏斜锁，并切换到轻量级锁实现。轻量级锁依赖 CAS 操作 Mark Word 来试图获取锁，如果重试成功，就使用普通的轻量级锁；否则，进一步升级为重量级锁（可能会先进行自旋锁升级，如果失败再尝试重量级锁升级）。

我注意到有的观点认为 Java 不会进行锁降级。实际上据我所知，锁降级确实是会发生的，当 JVM 进入安全点（SafePoint）的时候，会检查是否有闲置的 Monitor，然后试图进行降级。

2、Synchronized 优化后的锁机制简单介绍一下，包括自旋锁、偏向锁、轻量级锁、重量级锁？

自旋锁：

线程自旋说白了就是让 cpu 在做无用功，比如：可以执行几次 for 循环，可以执行几条空的汇编指令，目的是占着 CPU 不放，等待获取锁的机会。如果旋的时间过长会影响整体性能，时间过短又达不到延迟阻塞的目的。

偏向锁

偏向锁就是一旦线程第一次获得了监视对象，之后让监视对象“偏向”这个线程，之后的多次调用则可以避免 CAS 操作，说白了就是置个变量，如果发现为 true 则无需再走各种加锁/解锁流程。

轻量级锁：

轻量级锁是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁竞争用的时候，偏向锁就会升级为轻量级锁；

重量级锁

重量锁在 JVM 中又叫对象监视器（Monitor），它很像 C 中的 Mutex，除了具备 Mutex(0|1)互斥的功能，它还负责实现了 Semaphore(信号量)的功能，也就是说它至少包含一个竞争锁的队列，和一个信号阻塞队列（wait 队列），前者负责做互斥，后一个用于做线程同步。

3、谈谈对 Synchronized 关键字涉及到的类锁，方法锁，重入锁的理解？

synchronized 修饰静态方法获取的是类锁(类的字节码文件对象)。

synchronized 修饰普通方法或代码块获取的是对象锁。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态，从而有效避免了类成员变量的访问冲突。

它俩是不冲突的，也就是说：获取了类锁的线程和获取了对象锁的线程是不冲突的！

```
public class Widget {



    // 锁住了

    public synchronized void doSomething() {

        ...

    }

}



public class LoggingWidget extends Widget {



    // 锁住了

    public synchronized void doSomething() {
```

```
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
```

因为锁的持有者是“线程”，而不是“调用”。

线程 A 已经是有了 LoggingWidget 实例对象的锁了，当再需要的时候可以继续**“开锁”**进去的！

这就是内置锁的可重入性。

4、wait、sleep 的区别和 notify 运行过程。

wait、sleep 的区别

最大的不同是在等待时 wait 会释放锁，而 sleep 一直持有锁。wait 通常被用于线程间交互，sleep 通常被用于暂停执行。

- 首先，要记住这个差别，“sleep 是 Thread 类的方法,wait 是 Object 类中定义的方法”。尽管这两个方法都会影响线程的执行行为，但是本质上是有区别的。
- Thread.sleep 不会导致锁行为的改变，如果当前线程是拥有锁的，那么 Thread.sleep 不会让线程释放锁。如果能够帮助你记忆的话，可以简单认为和锁相关的方法都定义在 Object 类中，因此调用 Thread.sleep 是不会影响锁的相关行为。
- Thread.sleep 和 Object.wait 都会暂停当前的线程，对于 CPU 资源来说，不管是哪种方式暂停的线程，都表示它暂时不再需要 CPU 的执行时间。

OS 会将执行时间分配给其它线程。区别是，调用 wait 后，需要别的线程执行 notify/notifyAll 才能够重新获得 CPU 执行时间。

- 线程的状态参考 Thread.State 的定义。新创建的但是没有执行（还没有调用 start())的线程处于“就绪”，或者说 Thread.State.NEW 状态。
- Thread.State.BLOCKED（阻塞）表示线程正在获取锁时，因为锁不能获取到而被迫暂停执行下面的指令，一直等到这个锁被别的线程释放。BLOCKED 状态下线程，OS 调度机制需要决定下一个能够获取锁的线程是哪个，这种情况下，就是产生锁的争用，无论如何这都是很耗时的操作。

notify 运行过程

当线程 A（消费者）调用 wait()方法后，线程 A 让出锁，自己进入等待状态，同时加入锁对象的等待队列。 线程 B（生产者）获取锁后，调用 notify 方法通知锁对象的等待队列，使得线程 A 从等待队列进入阻塞队列。 线程 A 进入阻塞队列后，直至线程 B 释放锁后，线程 A 竞争得到锁继续从 wait()方法后执行。

5、synchronized 关键字和 Lock 的区别你知道吗？为什么 Lock 的性能好一些？

类别	synchronized	Lock（底层实现主要是 Volatile + CA
存在层次	Java 的关键字，在 jvm 层面上	是一个类
锁的释放	1、已获取锁的线程执行完同步代码，释放锁 2、线程执行发生异常，jvm 会让线程释放锁。	在 finally 中必须释放锁，不然容易造成线程
锁的获取	假设 A 线程获得锁，B 线程等待。如果 A 线程阻塞，B	分情况而定，Lock 有多个锁获取的方式，大

	线程会一直等待。	可以尝试获得锁，线程可以不用一直等待
锁状态	无法判断	可以判断
锁类型	可重入 不可中断 非公平	可重入 可判断 可公平（两者皆可）
性能	少量同步	大量同步

Lock（ReentrantLock）的底层实现主要是 Volatile + CAS（乐观锁），而

Synchronized 是一种悲观锁，比较耗性能。但是在 JDK1.6 以后对 Synchronized 的锁机制进行了优化，加入了偏向锁、轻量级锁、自旋锁、重量级锁，在并发量不大的情况下，性能可能优于 Lock 机制。所以建议一般请求并发量不大的情况

下使用 synchronized 关键字。

6、volatile 原理。

在《Java 并发编程：核心理论》一文中，我们已经提到可见性、有序性及原子性问题，通常情况下我们可以通过 Synchronized 关键字来解决这些个问题，不过

如果对 Synchonized 原理有了解的话，应该知道 Synchronized 是一个较重量级的操作，对系统的性能有比较大的影响，所以如果有其他解决方案，我们通常都避免使用 Synchronized 来解决问题。

而 volatile 关键字就是 Java 中提供的另一种解决可见性有序性问题的方案。对于原子性，需要强调一点，也是大家容易误解的一点：对 volatile 变量的单次读/写操作可保证原子性的，如 long 和 double 类型变量，但是并不能保证 i++这种操作的原子性，因为本质上 i++是读、写两次操作。

volatile 也是互斥同步的一种实现，不过它非常的轻量级。

volatile 的意义？

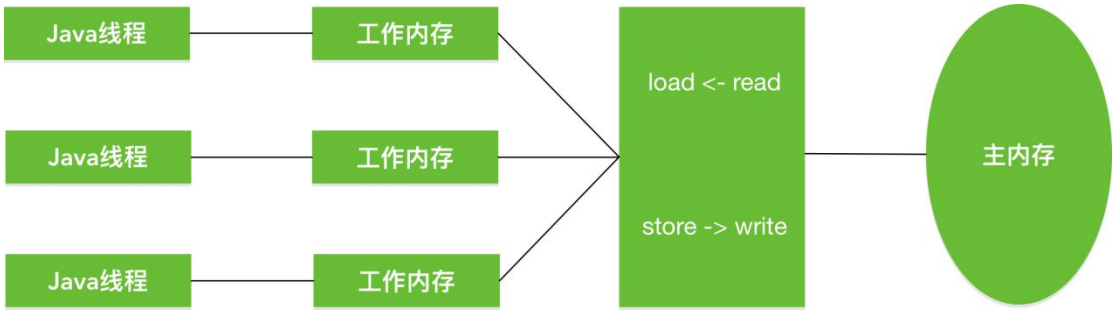
- 防止 CPU 指令重排序

volatile 有两条关键的语义：

保证被 volatile 修饰的变量对所有线程都是可见的

禁止进行指令重排序

要理解 volatile 关键字，我们得先从 Java 的线程模型开始说起。如图所示：



Java 内存模型规定了所有字段（这些字段包括实例字段、静态字段等，不包括局部变量、方法参数等，因为这些是线程私有的，并不存在竞争）都存在主内存中，每个线程会 有自己的工作内存，工作内存里保存了线程所使用到的变量在主内存里的副本拷贝，线程对变量的操作只能在工作内存里进行，而不能直接读写主内存，当然不同内存之间也 无法直接访问对方的工作内存，也就是说主内存是线程传值的媒介。

我们来理解第一句话：

```
保证被 volatile 修饰的变量对所有线程都是可见的
```

如何保证可见性？

被 volatile 修饰的变量在工作内存修改后会被强制写回主内存，其他线程在使用时也会强制从主内存刷新，这样就保证了一致性。

关于“保证被 volatile 修饰的变量对所有线程都是可见的”，有种常见的错误理解：

- 由于 volatile 修饰的变量在各个线程里都是一致的，所以基于 volatile 变量的运算在多线程并发的情况下是安全的。

这句话的前半部分是对的，后半部分却错了，因此它忘记考虑变量的操作是否具有原子性这一问题。

举个例子：

```
private volatile int start = 0;



private void volatile Keyword() {



    Runnable runnable = new Runnable() {

        @Override

        public void run() {

            for (int i = 0; i < 10; i++) {

                start++;

            }

        }

    };



    for (int i = 0; i < 10; i++) {

        Thread thread = new Thread(runnable);

        thread.start();

    }

    Log.d(TAG, "start = " + start);

}
```

```
09-20 16:21:27.471 1752-1752/com.guoxiaoxing.java.demo D/Thread: start = 50
```

这段代码启动了 10 个线程，每次 10 次自增，按道理最终结果应该是 100，但是结果并非如此。

为什么会这样？

仔细看一下 start++，它其实并非一个原子操作，简单来看，它有两步：

1、取出 start 的值，因为有 volatile 的修饰，这时候的值是正确的。

2、自增，但是自增的时候，别的线程可能已经把 start 加大了，这种情况下就有可能把较小的 start 写回主内存中。 所以 volatile 只能保证可见性，在不符合以下场景下我们依然需要通过加锁来保证原子性：

- 运算结果并不依赖变量当前的值，或者只有单一线程修改变量的值。（要么结果不依赖当前值，要么操作是原子性的，要么只要一个线程修改变量的值）
- 变量不需要与其他状态变量共同参与不变约束 比方说我们会在线程里加个 boolean 变量，来判断线程是否停止，这种情况就非常适合使用 volatile。

我们再来理解第二句话。

禁止进行指令重排序

什么是指令重排序？

> 指令重排序是指指令乱序执行，即在条件允许的情况下直接运行当前有能力立即执行的后续指令，避开为获取一条指令所需数据而造成的等待，通过乱序执行的技术提供执行效率。
>
> 指令重排序会在被 volatile 修饰的变量的赋值操作前，添加一个内存屏障，指令重排序时不能把后面的指令重排序移到内存屏障之前的位置。

7、synchronized 和 volatile 关键字的作用和区别。

Volatile

1）保证了不同线程对这个变量进行操作时的可见性即一个线程修改了某个变量的值，这新值对其他线程来是立即可见的。

2）禁止进行指令重排序。

作用

volatile 本质是在告诉 jvm 当前变量在寄存器（工作内存）中的值是不确定的，需从主存中读取；synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其它线程被阻塞住。

区别

1.volatile 仅能使用在变量级别；synchronized 则可以使用在变量、方法、和类级别的。

2.volatile 仅能实现变量的修改可见性，并不能保证原子性；synchronized 则可以保证变量的修改可见性和原子性。

3.volatile 不会造成线程的阻塞；synchronized 可能会造成线程的阻塞。

4.volatile 标记的变量不会被编译器优化；synchronized 标记的变量可以被编译器优化。

8、**ReentrantLock 的内部实现**。

ReentrantLock 实现的前提就是 AbstractQueuedSynchronizer，简称 AQS，是 java.util.concurrent 的核心，CountDownLatch、FutureTask、Semaphore、ReentrantLock 等都有一个内部类是这个抽象类的子类。由于 AQS 是基于 FIFO 队列的实现，因此必然存在一个个节点，Node 就是一个节点，Node 有两种模式：共享模式和独占模式。ReentrantLock 是基于 AQS 的，AQS 是 Java 并发包

中众多同步组件的构建基础，它通过一个 int 类型的状态变量 state 和一个 FIFO 队列来完成共享资源的获取，线程的排队等待等。AQS 是个底层框架，采用模板方法模式，它定义了通用的较为复杂的逻辑骨架，比如线程的排队，阻塞，唤醒等，将这些复杂但实质通用的部分抽取出来，这些都是需要构建同步组件的使用者无需关心的，使用者仅需重写一些简单的指定的方法即可（其实就是对于共享变量 state 的一些简单的获取释放的操作）。AQS 的子类一般只需要重写 tryAcquire(int arg)和 tryRelease(int arg)两个方法即可。

ReentrantLock 的处理逻辑：

其内部定义了三个重要的静态内部类，Sync，NonFairSync，FairSync。Sync 作为 ReentrantLock 中公用的同步组件，继承了 AQS（要利用 AQS 复杂的顶层逻辑嘛，线程排队，阻塞，唤醒等等）；NonFairSync 和 FairSync 则都继承 Sync，调用 Sync 的公用逻辑，然后再在各自内部完成自己特定的逻辑（公平或非公平）。

接着说下这两者的 lock()方法实现原理：

NonFairSync（非公平可重入锁）

1.先获取 state 值，若为 0，意味着此时没有线程获取到资源，CAS 将其设置为 1，设置成功则代表获取到排他锁了；

2.若 state 大于 0，肯定有线程已经抢占到资源了，此时再去判断是否就是自己抢占的，是的话，state 累加，返回 true，重入成功，state 的值即是线程重入的次数；

3.其他情况，则获取锁失败。

FairSync（公平可重入锁）

可以看到，公平锁的大致逻辑与非公平锁是一致的，不同的地方在于有了!hasQueuedPredecessors()这个判断逻辑，即便 state 为 0，也不能贸然直接去

获取，要先去看有没有还在排队的线程，若没有，才能尝试去获取，做后面的处理。反之，返回 false，获取失败。

最后，说下 ReentrantLock 的 tryRelease()方法实现原理：

若 state 值为 0，表示当前线程已完全释放干净，返回 true，上层的 AQS 会意识到资源已空出。若不为 0，则表示线程还占有资源，只不过将此次重入的资源的释放了而已，返回 false。

ReentrantLock 是一种可重入的，可实现公平性的互斥锁，它的设计基于 AQS 框架，可重入和公平性的实现逻辑都不难理解，每重入一次，state 就加 1，当然在释放的时候，也得一层一层释放。至于公平性，在尝试获取锁的时候多了一个判断：是否有比自己申请早的线程在同步队列中等待，若有，去等待；若没有，才允许去抢占。

9、ReentrantLock 、synchronized 和 volatile 比较?

synchronized 是互斥同步的一种实现。

synchronized：当某个线程访问被 synchronized 标记的方法或代码块时，这个线程便获得了该对象的锁，其他线暂时无法访问这个方法，只有等待这个方法执行完毕或代码块执行完毕，这个线程才会释放该对象的锁，其他线程才能执行这个方法代码块。

前面我们已经说了 volatile 关键字，这里我们举个例子来综合分析 volatile 与 synchronized 关键字的使用。

举个例子：

```
public class Singleton {



   // volatile 保证了：1 instance 在多线程并发的可见性 2 禁止 instance 在操作是的指令重排序
```

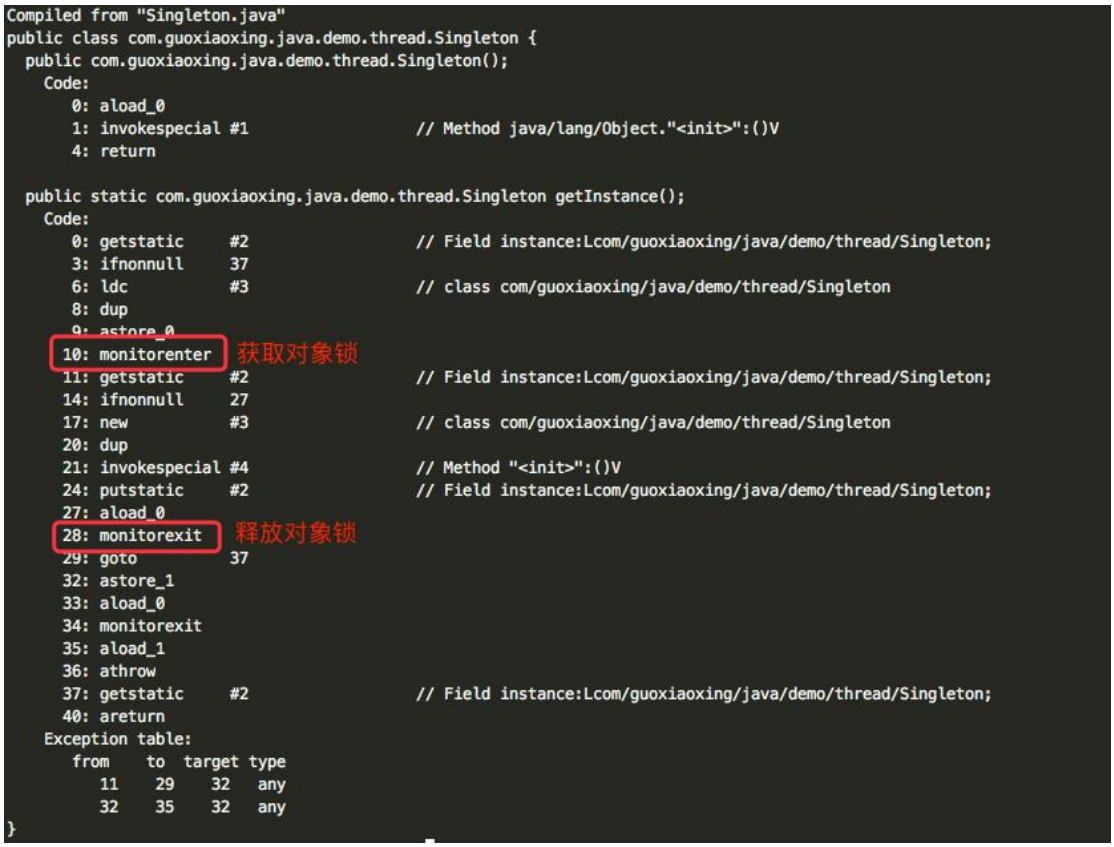
```
    private volatile static Singleton instance;


    private Singleton(){}


    public static Singleton getInstance() {
        // 第一次判空，保证不必要的同步
        if (instance == null) {
            // synchronized 对 Singleton 加全局锁，保证每次只要一个线程创建实例
            synchronized (Singleton.class) {
                // 第二次判空时为了在 null 的情况下创建实例
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

这是一个经典的 DCL 单例。

它的字节码如下：



可以看到被 synchronized 同步的代码块，会在前后分别加上 monitorenter 和 monitorexit，这两个字节码都需要指定加锁和解锁的对象。

关于加锁和解锁的对象：

synchronized 代码块 ：同步代码块，作用范围是整个代码块，作用对象是调用这个代码块的对象。

synchronized 方法 ：同步方法，作用范围是整个方法，作用对象是调用这个方法的对象。

synchronized 静态方法 ：同步静态方法，作用范围是整个静态方法，作用对象是调用这个类的所有对象。

synchronized(this)：作用范围是该对象中所有被 synchronized 标记的变量、方法或代码块，作用对象是对象本身。

synchronized(ClassName.class) ：作用范围是静态的方法或者静态变量，作用对象是 Class 对象。

synchronized(this)添加的是对象锁，synchronized(ClassName.class)添加的是类锁，它们的区别如下：

对象锁：Java 的所有对象都含有 1 个互斥锁，这个锁由 JVM 自动获取和释放。线程进入 synchronized 方法的时候获取该对象的锁，当然如果已经有线程获取了这个对象的锁那么当前线程会等待；synchronized 方法正常返回或者抛异常而终止，JVM 会自动释放对象锁。这里也体现了用 synchronized 来加锁的好处，方法抛异常的时候，锁仍然可以由 JVM 来自动释放。

类锁：对象锁是用来控制实例方法之间的同步，类锁是来控制静态方法（或静态变量互斥体）之间的同步。其实类锁只是一个概念上的东西，并不是真实存在的，它只用来帮助我们理解锁定实例方法和静态方法的区别的。我们都知道，java 类可能会有很多个对象，但是只有 1 个 Class 对象，也就说类的不同实例之间共享该类的 Class 对象。Class 对象其实也仅仅是 1 个 java 对象，只不过有点特殊而已。由于每个 java 对象都有个互斥锁，而类的静态方法是需要 Class 对象。所以所谓类锁，不过是 Class 对象的锁而已。获取类的 Class 对象有好几种，最简单的就是 MyClass.class 的方式。类锁和对象锁不是同一个东西，一个是类的 Class 对象的锁，一个是类的实例的锁。也就是说：一个线程访问静态 sychronized 的时候，允许另一个线程访问对象的实例 synchronized 方法。反过来也是成立的，为他们需要的锁是不同的。

-

三、其它 （★ ★ ★ ）

1、多线程的使用场景？

使用多线程就一定效率高吗？有时候使用多线程并不是为了提高效率，而是使得 CPU 能同时处理多个事件。

为了不阻塞主线程,启动其他线程来做事情,比如 APP 中的耗时操作都不在 UI 线程中做。

实现更快的应用程序,即主线程专门监听用户请求,子线程用来处理用户请求,以获得大的吞吐量.感觉这种情况，多线程的效率未必高。这种情况下的多线程是为了不必等待，可以并行处理多条数据。比如 JavaWeb 的就是主线程专门监听用户的 HTTP 请求，然启动子线程去处理用户的 HTTP 请求。

某种虽然优先级很低的服务，但是却要不定时去做。比如 Jvm 的垃圾回收。

某种任务，虽然耗时，但是不消耗 CPU 的操作时间，开启个线程，效率会有显著提高。比如读取文件，然后处理。磁盘 IO 是个很耗费时间，但是不耗 CPU 计算的工作。所以可以一个线程读取数据，一个线程处理数据。肯定比一个线程读取数据，然后处理效率高。因为两个线程的时候充分利用了 CPU 等待磁盘 IO 的空闲时间。

2、CopyOnWriteArrayList 的了解。

Copy-On-Write 是什么？

在计算机中就是当你想要对一块内存进行修改时，我们不在原有内存块中进行写操作，而是将内存拷贝一份，在新的内存中进行写操作，写完之后呢，就将指向原来内存指针指向新的内存，原来的内存就可以被回收掉。

原理：

CopyOnWriteArrayList 这是一个 ArrayList 的线程安全的变体，

CopyOnWriteArrayList 底层实现添加的原理是先 copy 出一个容器(可以简称副本)，再往新的容器里添加这个新的数据，最后把新的容器的引用地址赋值给了之前那个旧的的容器地址，但是在添加这个数据的期间，其他线程如果要去读取数据，仍然是读取到旧的容器里的数据。

优点和缺点:

优点:

1.据一致性完整，为什么？因为加锁了，并发数据不会乱。

2.解决了像 ArrayList、Vector 这种集合多线程遍历迭代问题，记住，Vector 虽然线程安全，只不过是加了 synchronized 关键字，迭代问题完全没有解决！

缺点:

1.内存占有问题:很明显，两个数组同时驻扎在内存中，如果实际应用中，数据比较多，而且比较大的情况下，占用内存会比较大，针对这个其实可以用 ConcurrentHashMap 来代替。

2.数据一致性:CopyOnWrite 容器只能保证数据的最终一致性，不能保证数据的实时一致性。所以如果你希望写入的的数据，马上能读到，请不要使用 CopyOnWrite 容器。

使用场景：

1、读多写少（白名单，黑名单，商品类目的访问和更新场景），为什么？因为写的时候会复制新集合。

2、集合不大，为什么？因为写的时候会复制新集合。

3、实时性要求不高，为什么，因为有可能会读取到旧的集合数据。

3、ConcurrentHashMap 加锁机制是什么，详细说一下？

Java7 ConcurrentHashMap

ConcurrentHashMap 作为一种线程安全且高效的哈希表的解决方案，尤其其中的"分段锁"的方案，相比 HashTable 的表锁在性能上的提升非常之大。HashTable 容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问 HashTable 的线程都必须竞争同一把锁，那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效的提高并发访问效率，这就是 ConcurrentHashMap 所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

concurrencyLevel：并行级别、并发数、Segment 数。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。其中的每个 Segment 很像 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

初始化槽: ensureSegment

ConcurrentHashMap 初始化的时候会初始化第一个槽 segment[0]，对于其他槽来说，在插入第一个值的时候进行初始化。对于并发操作使用 CAS 进行控制。

Java8 ConcurrentHashMap

抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。结构上和 Java8 的 HashMap（数组+链表+红黑树） 基本上一样，不过它要保证线程安全性，所以在源码上确实要复杂一些。1.8 在 1.7 的数据结构上做了大的改动，采用红黑树之后可以保证查询效率（O(logn)），甚至取消了 ReentrantLock 改为了 synchronized，这样可以看出在新版的 JDK 中对 synchronized 优化是很到位的。

4、线程死锁的 4 个条件？

死锁是如何发生的，如何避免死锁？

当线程 A 持有独占锁 a，并尝试去获取独占锁 b 的同时，线程 B 持有独占锁 b，并尝试获取独占锁 a 的情况下，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。

```
public class DeadLockDemo {



   public static void main(String[] args) {
       // 线程 a
       Thread td1 = new Thread(new Runnable() {
           public void run() {
               DeadLockDemo.method1();
           }
       });
```

```
        // 线程 b
        Thread td2 = new Thread(new Runnable() {
            public void run() {
                DeadLockDemo.method2();
            }
        });

        td1.start();
        td2.start();
    }

    public static void method1() {
        synchronized (String.class) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("线程 a 尝试获取 integer.class");
            synchronized (Integer.class) {

            }
        }
    }
```

```
    public static void method2() {

        synchronized (Integer.class) {

            try {

                Thread.sleep(2000);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

            System.out.println("线程 b 尝试获取 String.class");

            synchronized (String.class) {



            }

        }

    }

}
```

造成死锁的四个条件：

- 互斥条件：一个资源每次只能被一个线程使用。
- 请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：线程已获得的资源，在未使用完之前，不能强行剥夺。
- 循环等待条件：若干线程之间形成一种头尾相接的循环等待资源关系。

在并发程序中，避免了逻辑中出现数个线程互相持有对方线程所需要的独占锁的的情况，就可以避免死锁，如下所示：

```
public class BreakDeadLockDemo {



    public static void main(String[] args) {

        // 线程 a

        Thread td1 = new Thread(new Runnable() {
```

```
            public void run() {
                DeadLockDemo2.method1();
            }
        });
        // 线程 b
        Thread td2 = new Thread(new Runnable() {
            public void run() {
                DeadLockDemo2.method2();
            }
        });

        td1.start();
        td2.start();
    }

    public static void method1() {
        synchronized (String.class) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("线程 a 尝试获取 integer.class");
            synchronized (Integer.class) {
                System.out.println("线程 a 获取到 integer.class");
```

```
            }


        }
    }


    public static void method2() {
        // 不再获取线程 a 需要的 Integer.class 锁。
        synchronized (String.class) {
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("线程 b 尝试获取 Integer.class");
            synchronized (Integer.class) {
                System.out.println("线程 b 获取到 Integer.class");
            }
        }
    }
}
```

5、CAS 介绍？

Unsafe

Unsafe 是 CAS 的核心类。因为 Java 无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM 还是开了一个后门，JDK 中有一个类 Unsafe，它提供了硬件级别的原子操作。

CAS

CAS，Compare and Swap 即比较并交换，设计并发算法时常用到的一种技术，java.util.concurrent 包全完建立在 CAS 之上，没有 CAS 也就没有此包，可见 CAS 的重要性。当前的处理器基本都支持 CAS，只不过不同的厂家的实现不一样罢了。并且 CAS 也是通过 Unsafe 实现的，由于 CAS 都是硬件级别的操作，因此效率会比普通加锁高一些。

CAS 的缺点

CAS 看起来很美，但这种操作显然无法涵盖并发下的所有场景，并且 CAS 从语义上来说也不是完美的，存在这样一个逻辑漏洞：如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然是 A 值，那我们就能说明它的值没有被其他线程修改过了吗？如果在这段期间它的值曾经被改成了 B，然后又改回 A，那 CAS 操作就会误认为它从来没有被修改过。这个漏洞称为 CAS 操作的"ABA"问题。java.util.concurrent 包为了解决这个问题，提供了一个带有标记的原子引用类"AtomicStampedReference"，它可以通过控制变量值的版本来保证 CAS 的正确性。不过目前来说这个类比较"鸡肋"，大部分情况下 ABA 问题并不会影响程序并发的正确性，如果需要解决 ABA 问题，使用传统的互斥同步可能回避原子类更加高效。

6、进程和线程的区别？

简而言之,一个程序至少有一个进程,一个进程至少有一个线程。

1、线程的划分尺度小于进程，使得多线程程序的并发性高。

2、进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

3、线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

4、从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

5、进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

6、一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行。

7、进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。

7、什么导致线程阻塞？

线程的阻塞

为了解决对共享存储区的访问冲突，Java 引入了同步机制，现在让我们来考察多个线程对共享资源的访问，显然同步机制已经不够了，因为在任意时刻所要求的资源不一定已经准备好了被访问，反过来，同一时刻准备好了的资源也可能不止一个。为了解决这种情况下的访问控制问题，Java 引入了对阻塞机制的支持.

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。

sleep() 方法：sleep() 允许 指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新进入可执行状态。 典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

suspend() 和 resume() 方法：两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。

yield() 方法：yield() 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。

wait() 和 notify() 方法：两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用。初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙

述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。

上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的 notify()方法则导致因调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在 synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现 IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和 synchronized 方法或块一起使用，将它们和操作系统的进程间通信机制作一个比较就会发现它们的相似性：synchronized 方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和 wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。（此外，线程间通信的方式还有多个线程通过 synchronized 关键字这种方式来实现线程间的通信、while 轮询、使用 java.io.PipedInputStream 和 java.io.PipedOutputStream 进行通信的管道通信）。

关于 wait() 和 notify() 方法最后再说明两点：

第一：调用 notify() 方法导致解除阻塞的线程是从调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

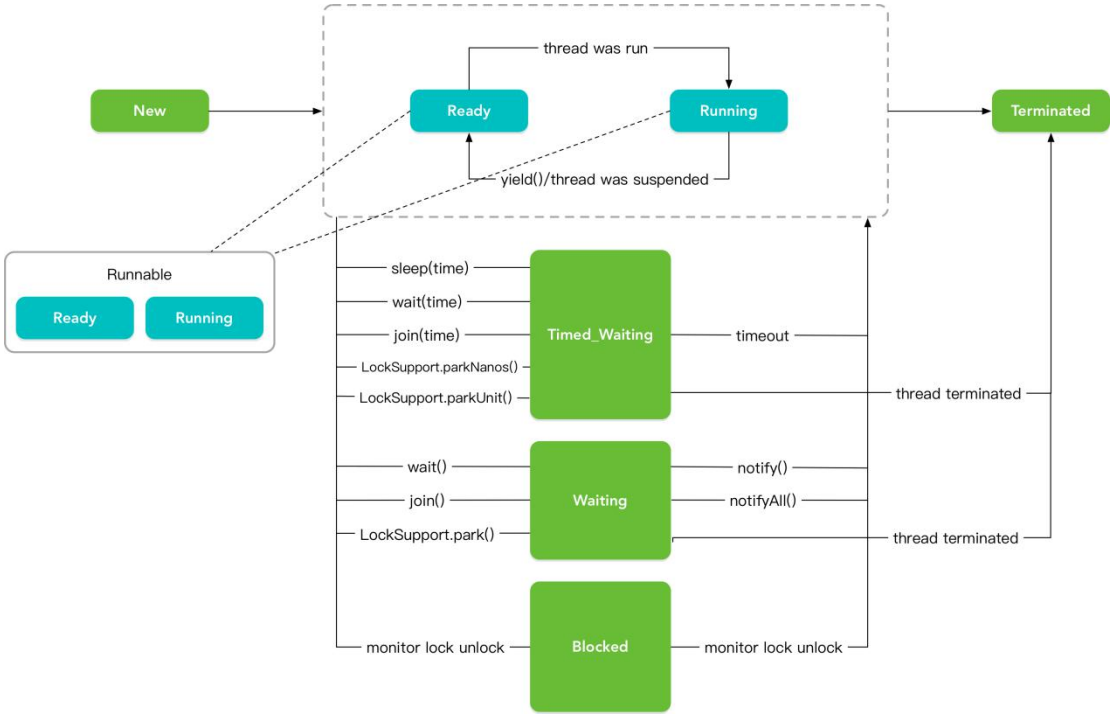
第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

8、线程的生命周期

线程状态流程图



- NEW：创建状态，线程创建之后，但是还未启动。
- RUNNABLE：运行状态，处于运行状态的线程，但有可能处于等待状态，例如等待 CPU、IO 等。
- WAITING：等待状态，一般是调用了 wait()、join()、LockSupport.spark()等方法。
- TIMED_WAITING：超时等待状态，也就是带时间的等待状态。一般是调用了 wait(time) 、 join(time) 、 LockSupport.sparkNanos() 、 LockSupport.sparkUnit()等方法。
- BLOCKED：阻塞状态，等待锁的释放，例如调用了 synchronized 增加了锁。
- TERMINATED：终止状态，一般是线程完成任务后退出或者异常终止。

NEW、WAITING、TIMED_WAITING 都比较好理解，我们重点说一说 RUNNABLE 运行态和 BLOCKED 阻塞态。

线程进入 RUNNABLE 运行态一般分为五种情况：

- 线程调用 sleep(time)后结束了休眠时间
- 线程调用的阻塞 IO 已经返回，阻塞方法执行完毕
- 线程成功的获取了资源锁
- 线程正在等待某个通知，成功的获得了其他线程发出的通知
- 线程处于挂起状态，然后调用了 resume()恢复方法，解除了挂起。

线程进入 BLOCKED 阻塞态一般也分为五种情况：

- 线程调用 sleep()方法主动放弃占有的资源
- 线程调用了阻塞式 IO 的方法，在该方法返回前，该线程被阻塞。
- 线程视图获得一个资源锁，但是该资源锁正被其他线程锁持有。
- 线程正在等待某个通知
- 线程调度器调用 suspend()方法将该线程挂起

我们再来看看和线程状态相关的一些方法。

sleep()方法让当前正在执行的线程在指定时间内暂停执行，正在执行的线程可以通过 Thread.currentThread()方法获取。

yield()方法放弃线程持有的 CPU 资源，将其让给其他任务去占用 CPU 执行时间。但放弃的时间不确定，有可能刚刚放弃，马上又获得 CPU 时间片。

wait()方法是当前执行代码的线程进行等待，将当前线程放入预执行队列，并在 wait()所在的代码处停止执行，直到接到通知或者被中断为止。该方法可以使得调用该方法的线程释放共享资源的锁，然后从运行状态退出，进入等待队列，直到再次被唤醒。该方法只能在同步代码块里调用，否则会抛出 IllegalMonitorStateException 异常。wait(long millis)方法等待某一段时间内是否有线程对锁进行唤醒，如果超过了这个时间则自动唤醒。

notify()方法用来通知那些可能等待该对象的对象锁的其他线程，该方法可以随机唤醒等待队列中等同一共享资源的一个线程，并使该线程退出等待队列，进入可运行状态。

notifyAll()方法可以使所有正在等待队列中等待同一共享资源的全部线程从等待状态退出，进入可运行状态，一般会是优先级高的线程先执行，但是根据虚拟机的实现不同，也有可能是随机执行。

join()方法可以让调用它的线程正常执行完成后，再去执行该线程后面的代码，它具有让线程排队的作用。

9、乐观锁与悲观锁。

悲观锁

总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程）。Java 中 synchronized 和 ReentrantLock 等独占锁就是悲观锁思想的实现。

乐观锁

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和 CAS 算法实现。乐观锁适用于多读的应用类型，这样可以提高吞吐量。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

使用场景

乐观锁适用于写比较少的情况下（多读场景），而一般多写的场景下用悲观锁就比较合适。

乐观锁常见的两种实现方式

1、版本号机制

一般是在数据表中加上一个数据版本号 version 字段，表示数据被修改的次数，当数据被修改时，version 值会加 1。当线程 A 要更新数据值时，在读取数据的同时也会读取 version 值，在提交更新时，若刚才读取到的 version 值为当前数据库中的 version 值相等时才更新，否则重试更新操作，直到更新成功。

2、CAS 算法

即 compare and swap（比较与交换），是一种有名的无锁算法。CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则什么都不做。 一般情况下是一个自旋操作，即不断的重试。

乐观锁的缺点

1、ABA 问题

如果一个变量 V 初次读取的时候是 A 值，并且在准备赋值的时候检查到它仍然是 A 值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回 A，那 CAS 操作就会误认为它从来没有被修改过。这个问题被称为 CAS 操作的 "ABA"问题。

JDK 1.5 以后的 AtomicStampedReference 类一定程度上解决了这个问题，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

2、自旋 CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给 CPU 带来非常大的执行开销。

3、CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5 开始，提供了 AtomicReference 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用 AtomicReference 类把多个共享变量合并成一个共享变量来操作。

10、run()和 start()方法区别？

1.start()方法来启动线程，真正实现了多线程运行，这时无需等待 run 方法体代码执行完毕而直接继续执行下面的代码：

通过调用 Thread 类的 start()方法来启动一个线程，这时此线程是处于就绪状态，并没有运行。 然后通过此 Thread 类调用方法 run()来完成其运行操作的， 这里方法 run()称为线程体， 它包含了要执行的这个线程的内容， Run 方法运行结束， 此线程终止， 而 CPU 再运行其它线程，在 Android 中一般是主线程。

2.run()方法当作普通方法的方式调用，程序还是要顺序执行，还是要等待 run 方法体执行完毕后才可继续执行下面的代码：

而如果直接用 Run 方法， 这只是调用一个方法而已， 程序中依然只有主线程--这一个线程，其程序执行路径还是只有一条，这样就没有达到写线程的目的。

11、多线程断点续传原理。

在本地下载过程中要使用数据库实时存储到底存储到文件的哪个位置了，这样点击开始继续传递时，才能通过 HTTP 的 GET 请求中的 setRequestProperty("Range","bytes=startIndex-endIndex");方法可以告诉服务器，数据从哪里开始，到哪里结束。同时在本地的文件写入时，RandomAccessFile 的 seek()方法也支持在文件中的任意位置进行写入操作。同时通过广播或事件总线机制将子线程的进度告诉 Activity 的进度条。关于断线续传的 HTTP 状态码是 206，即 HttpStatus.SC_PARTIAL_CONTENT。

12、怎么安全停止一个线程任务？原理是什么？线程池里有类似机制吗？

终止线程

1、使用 violate boolean 变量退出标志，使线程正常退出，也就是当 run 方法完成后线程终止。（推荐）

2、使用 interrupt()方法中断线程，但是线程不一定会终止。

3、使用 stop 方法强行终止线程。不安全主要是：thread.stop()调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子线程所持有的所有锁。

终止线程池

ExecutorService 线程池就提供了 shutdown 和 shutdownNow 这样的生命周期方法来关闭线程池自身以及它拥有的所有线程。

1、shutdown 关闭线程池

线程池不会立刻退出，直到添加到线程池中的任务都已经处理完成，才会退出。

2、shutdownNow 关闭线程池并中断任务

终止等待执行的线程，并返回它们的列表。试图停止所有正在执行的线程，试图终止的方法是调用 Thread.interrupt()，但是大家知道，如果线程中没有 sleep 、wait、Condition、定时锁等应用, interrupt()方法是无法中断当前的线程的。所以，ShutdownNow()并不代表线程池就一定立即就能退出，它可能必须要等待所有正在执行的任务都执行完成了才能退出。

13、堆内存，栈内存理解，栈如何转换成堆？

- 在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配。
- 堆内存用于存放由 new 创建的对象和数组。JVM 里的“堆”（heap）特指用于存放 Java 对象的内存区域。所以根据这个定义，Java 对象全部都在堆上。JVM 的堆被同一个 JVM 实例中的所有 Java 线程共享。它通常由某种自动内存管理机制所管理，这种机制通常叫做“垃圾回收”（garbage collection，GC）。
- 堆主要用来存放对象的，栈主要是用来执行程序的。
- 实际上，栈中的变量指向堆内存中的变量，这就是 Java 中的指针!

14、如何控制某个方法允许并发访问线程的个数；

15、多进程开发以及多进程应用场景；

16、Java 的线程模型；

17、死锁的概念，怎么避免死锁？

18、如何保证多线程读写文件的安全？

19、线程如何关闭，以及如何防止线程的内存泄漏？

20、为什么要有线程，而不是仅仅用进程？

21、多个线程如何同时请求，返回的结果如何等待所有线程数据完成后合成一个数据？

22、线程如何关闭？

23、数据一致性如何保证？

24、两个进程同时要求写或者读，能不能实现？如何防止进程的同步？

25、谈谈对多线程的理解并举例说明

26、线程的状态和优先级。

27、ThreadLocal 的使用

28、Java 中的并发工具（CountDownLatch，CyclicBarrier 等）

29、进程线程在操作系统中的实现

30、双线程通过线程同步的方式打印 12121212.....

31、java 线程，场景实现，多个线程如何同时请求，返回的结果如何等待所有线程数据完成后合成一个数据

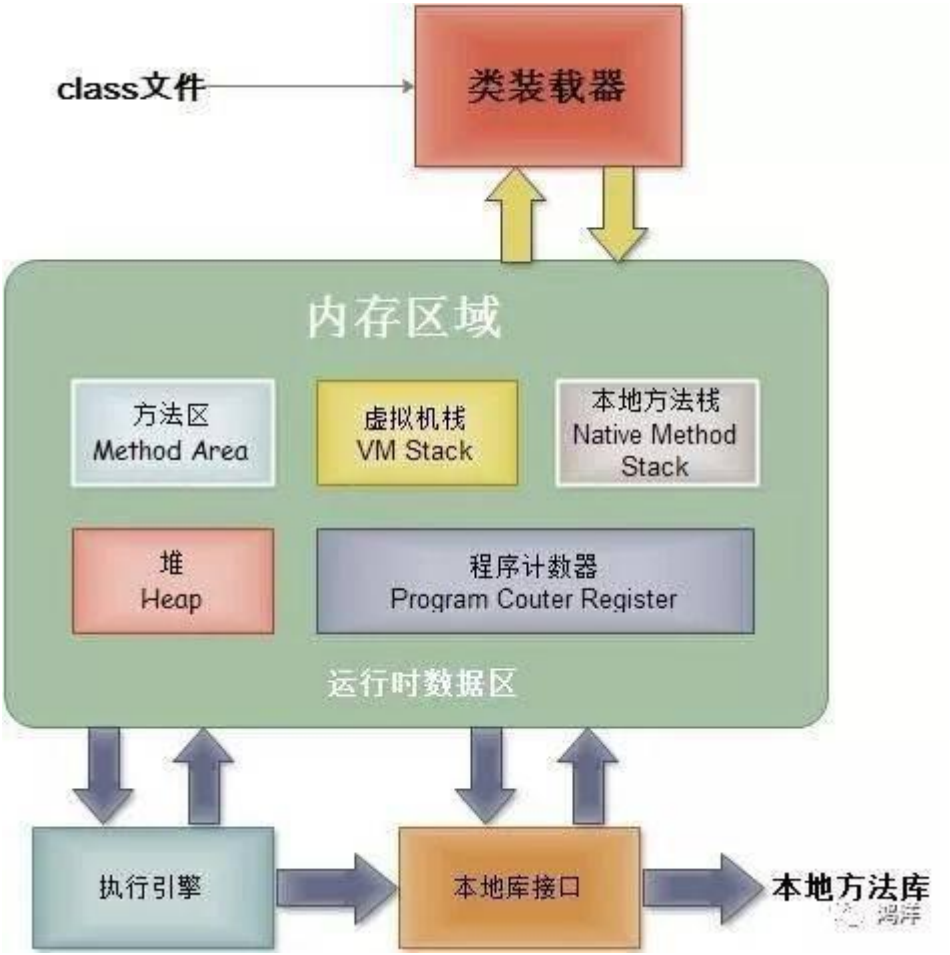
32、服务器只提供数据接收接口，在多线程或多进程条件下，如何保证数据的有序到达？

33、单机上一个线程池正在处理服务，如果忽然断电了怎么办（正在处理和阻塞队列里的请求怎么处理）？

第三节 Java 虚拟机面试题 （★ ★ ★）

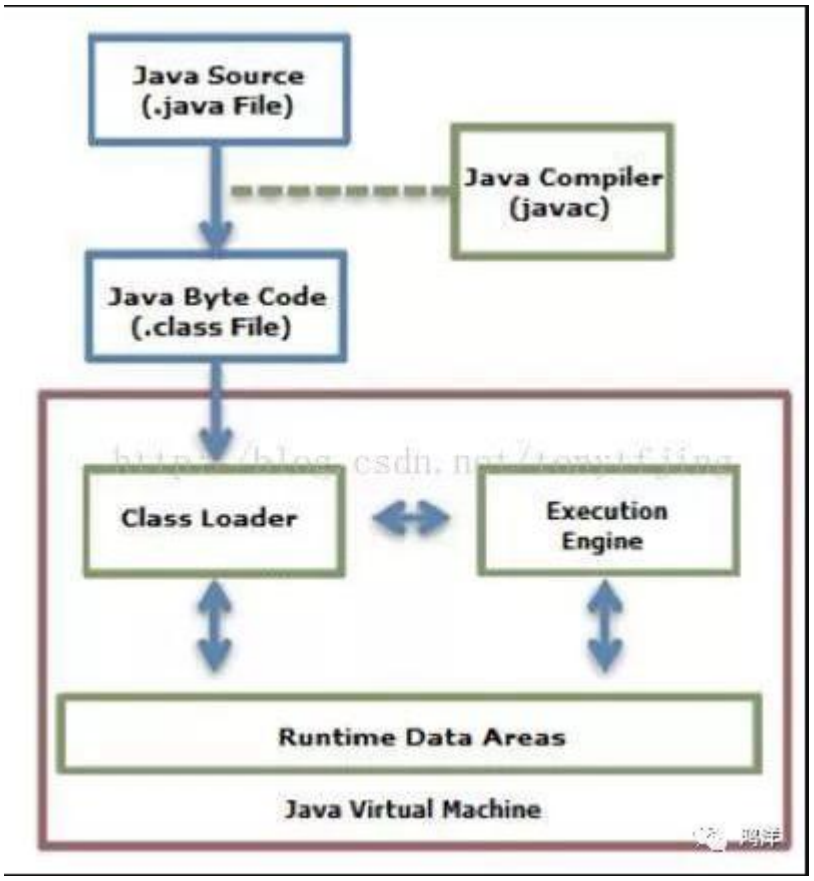
1、JVM 内存区域。

JVM 基本构成



从上图可知，JVM 主要包括四个部分：

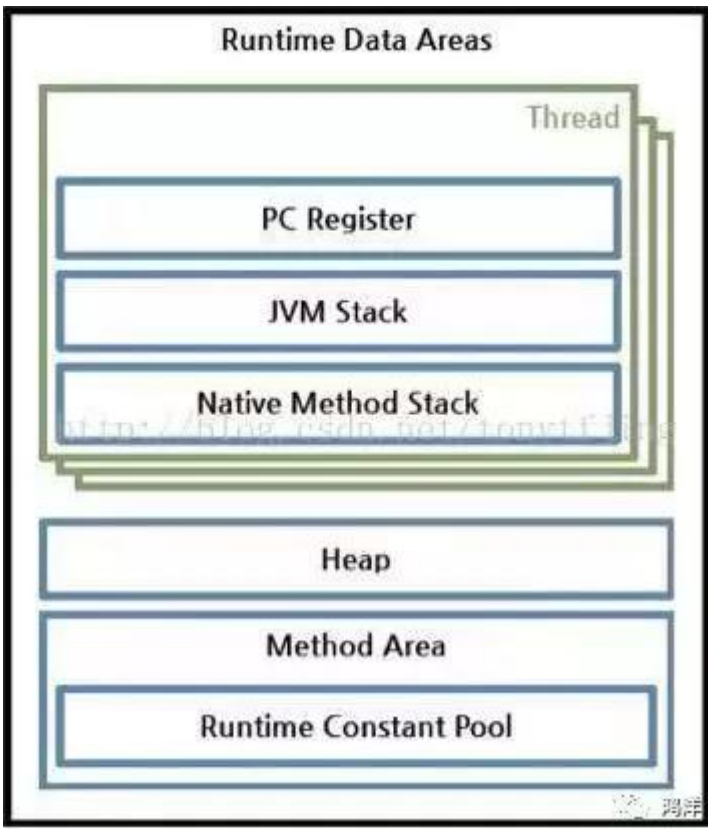
1.类加载器（ClassLoader）:在 JVM 启动时或者在类运行将需要的 class 加载到 JVM 中。（下图表示了从 java 源文件到 JVM 的整个过程，可配合理解。



2.执行引擎：负责执行 class 文件中包含的字节码指令；

3.内存区（也叫运行时数据区）：是在 JVM 运行的时候操作所分配的内存区。

运行时内存区主要可以划分为 5 个区域，如图：



方法区(MethodArea)：用于存储类结构信息的地方，包括常量池、静态常量、构造函数等。虽然 JVM 规范把方法区描述为堆的一个辑部分， 但它却有个别名 non-heap（非堆），所以大家不要搞混淆了。方法区还包含一个运行时常量池。

java 堆(Heap)：存储 java 实例或者对象的地方。这块是 GC 的主要区域。从存储的内容我们可以很容易知道，方法和堆是被所有 java 线程共享的。

java 栈(Stack)：java 栈总是和线程关联在一起，每当创一个线程时，JVM 就会为这个线程创建一个对应的 java 栈在这个 java 栈中,其中又会包含多个栈帧，每运行一个方法就建一个栈帧，用于存储局部变量表、操作栈、方法返回等。每一个方法从调用直至执行完成的过程，就对应一栈帧在 java 栈中入栈到出栈的过程。所以 java 栈是现成有的。

程序计数器(PCRegister)：用于保存当前线程执行的内存地址。由于 JVM 程序是多线程执行的（线程轮流切换），所以为了保证程切换回来后，还能恢复到原先状态，就需要一个独立计数器，记录之前中断的地方，可见程序计数器也是线程私有的。

本地方法栈(Native MethodStack)：和 java 栈的作用差不多，只不过是为 JVM 使用到 native 方法服务的。

4.本地方法接口：主要是调用 C 或 C++实现的本地方法及回调结果。

开线程影响哪块内存？

每当有线程被创建的时候，JVM 就需要为其在内存中分配虚拟机栈和本地方法栈来记录调用方法的内容，分配程序计数器记录指令执行的位置，这样的内存消耗就是创建线程的内存代价。

2、JVM 的内存模型的理解？

Java 内存模型即 Java Memory Model，简称 JMM。JMM 定义了 Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM 是整个计算机虚拟模型，所以 JMM 是隶属于 JVM 的。

Java 线程之间的通信总是隐式进行，并且采用的是共享内存模型。这里提到的共享内存模型指的就是 Java 内存模型(简称 JMM)，JMM 决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM 定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

总之，JMM 就是一组规则，这组规则意在解决在并发编程可能出现的线程安全问题，并提供了内置解决方案（happen-before 原则）及其外部可使用的同步手段(synchronized/volatile 等)，确保了程序执行在多线程环境中的应有的原子性，可视性及其有序性。

需要更全面理解建议阅读以下文章：

全面理解 Java 内存模型(JMM)及 volatile 关键字

全面理解 Java 内存模型

3、描述一下 GC 的原理和回收策略？

提到垃圾回收，我们可以先思考一下，如果我们去做垃圾回收需要解决哪些问题？

一般说来，我们要解决三个问题：

1、回收哪些内存？

2、什么时候回收？

3、如何回收？

这些问题分别对应着引用管理和回收策略等方案。

提到引用，我们都知道 Java 中有四种引用类型：

- 强引用：代码中普遍存在的，只要强引用还存在，垃圾收集器就不会回收掉被引用的对象。
- 软引用：SoftReference，用来描述还有用但是非必须的对象，当内存不足的时候会回收这类对象。

- 弱引用：WeakReference，用来描述非必须对象，弱引用的对象只能生存到下一次 GC 发生时，当 GC 发生时，无论内存是否足够，都会回收该对象。
- 虚引用：PhantomReference，一个对象是否有虚引用的存在，完全不会对其生存时间产生影响，也无法通过虚引用取得一个对象的引用，它存在的唯一目的是在这个对象被回收时可以收到一个系统通知。

不同的引用类型，在做 GC 时会区别对待，我们平时生成的 Java 对象，默认都是强引用，也就是说只要强引用还在，GC 就不会回收，那么如何判断强引用是否存在呢？

一个简单的思路就是：引用计数法，有对这个对象的引用就+1，不再引用就-1，但是这种方式看起来简单美好，但它却不能解决循环引用计数的问题。

因此可达性分析算法登上历史舞台，用它来判断对象的引用是否存在。

可达性分析算法通过一系列称为 GCRoots 的对象作为起始点，从这些节点从上向下搜索，所走过的路径称为引用链，当一个对象没有任何引用链与 GCRoots 连接时就说明此对象不可用，也就是对象不可达。

GC Roots 对象通常包括：

- 虚拟机栈中引用的对象（栈帧中的本地变量表）
- 方法中类的静态属性引用的对象
- 方法区中常量引用的对象
- Native 方法引用的对象

可达性分析算法整个流程如下所示：

第一次标记：对象在经过可达性分析后发现没有与 GC Roots 有引用链，则进行第一次标记并进行一次筛选，筛选条件是：该对象是否有必要执行 finalize()方法。

没有覆盖 finalize()方法或者 finalize()方法已经被执行过都会被认为没有必要执

行。 如果有必要执行：则该对象会被放在一个 F-Queue 队列，并稍后在由虚拟机建立的低优先级 Finalizer 线程中触发该对象的 finalize()方法，但不保证一定等待它执行结束，因为如果这个对象的 finalize()方法发生了死循环或者执行时间较长的情况，会阻塞 F-Queue 队列里的其他对象，影响 GC。

第二次标记：GC 对 F-Queue 队列里的对象进行第二次标记，如果在第二次标记时该对象又成功被引用，则会被移除即将回收的集合，否则会被回收。

总之，JVM 在做垃圾回收的时候，会检查堆中的所有对象否会被这些根集对象引用，不能够被引用的对象就会被圾收集器回收。一般回收算法也有如下几种：

1).标记-清除（Mark-sweep）

标记-清除算法采用从根集合进行扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收。标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。

2).标记-整理（Mark-Compact）

标记-整理算法采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。标记-整理算法是在标记-清除算法的基础上，又进行了对象的移动，因此成本更高，但是却解决了内存碎片的问题。该垃圾回收算法适用于对象存活率高的场景（老年代）。

3).复制（Copying）

复制算法将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这种算法适用于对象存活率低的场景，比如新生

代。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况。

4).分代收集算法

不同的对象的生命周期(存活情况)是不一样的，而不同生命周期的对象位于堆中不同的区域，因此对堆内存不同区域采用不同的策略进行回收可以提高 JVM 的执行效率。当代商用虚拟机使用的都是分代收集算法：新生代对象存活率低，就采用复制算法；老年代存活率高，就用标记清除算法或者标记整理算法。Java 堆内存一般可以分为新生代、老年代和永久代三个模块：

新生代：

1.所有新生成的对象首先都是放在新生代的。新生代的目标就是尽可能快速的收集掉那些生命周期短的对象。

2.新生代内存按照 8:1:1 的比例分为一个 eden 区和两个 survivor(survivor0,survivor1)区。大部分对象在 Eden 区中生成。回收时先将 eden 区存活对象复制到一个 survivor0 区，然后清空 eden 区，当这个 survivor0 区也存放满了时，则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区，然后清空 eden 和这个 survivor0 区，此时 survivor0 区是空的，然后将 survivor0 区和 survivor1 区交换，即保持 survivor1 区为空， 如此往复。

3.当 survivor1 区不足以存放 eden 和 survivor0 的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次 Full GC，也就是新生代、老年代都进行回收。

4.新生代发生的 GC 也叫做 Minor GC，MinorGC 发生频率比较高(不一定等 Eden 区满了才触发)。

老年代：

1.在老年代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到老年代中。因此，可以认为老年代中存放的都是一些生命周期较长的对象。

2.内存比新生代也大很多(大概比例是 1:2)，当老年代内存满时触发 Major GC，即 Full GC。Full GC 发生频率比较低，老年代对象存活时间比较长。

永久代：

永久代主要存放静态文件，如 Java 类、方法等。永久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如使用反射、动态代理、CGLib 等 bytecode 框架时，在这种时候需要设置一个比较大的永久代空间来存放这些运行过程中新增的类。

垃圾收集器

垃圾收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现：

Serial 收集器（复制算法): 新生代单线程收集器，标记和清理都是单线程，优点是简单高效；

Serial Old 收集器 (标记-整理算法): 老年代单线程收集器，Serial 收集器的老年代版本；

ParNew 收集器 (复制算法): 新生代收并行集器，实际上是 Serial 收集器的多线程版本，在多核 CPU 环境下有着比 Serial 更好的表现；

CMS(Concurrent Mark Sweep)收集器（标记-清除算法）： 老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短 GC 回收停顿时间。

Parallel Old 收集器 (标记-整理算法)： 老年代并行收集器，吞吐量优先，Parallel Scavenge 收集器的老年代版本；

Parallel Scavenge 收集器 (复制算法): 新生代并行收集器，追求高吞吐量，高效利用 CPU。吞吐量 = 用户线程时间/(用户线程时间+GC 线程时间)，高吞吐量可以高效率的利用 CPU 时间，尽快完成程序的运算任务，适合后台应用等对交互相应要求不高的场景；

G1(Garbage First)收集器 (标记-整理算法)： Java 堆并行收集器，G1 收集器是 JDK1.7 提供的一个新收集器，G1 收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1 收集器不同于之前的收集器的一个重要特点是：G1 回收的范围是整个 Java 堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。

内存分配和回收策略

JAVA 自动内存管理：给对象分配内存 以及 回收分配给对象的内存。

1、对象优先在 Eden 分配，当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 MinorGC。

2、大对象直接进入老年代。如很长的字符串以及数组。很长的字符串以及数组。

3、长期存活的对象将进入老年代。当对象在新生代中经历过一定次数（默认为 15）的 Minor GC 后，就会被晋升到老年代中。

4、动态对象年龄判定。为了更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象年龄必须达到了 MaxTenuringThreshold 才能晋升老年代，如果在

Survivor 空间中相同年龄所有对象大小的总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到 MaxTenuringThreshold 中要求的年龄。

需要更全面的理解请点击这里

4、类的加载器，双亲机制，Android 的类加载器。

类的加载器

大家都知道，一个 Java 程序都是由若干个.class 文件组织而成的一个完整的 Java 应用程序，当程序在运行时，即会调用该程序的一个入口函数来调用系统的相关功能，而这些功能都被封装在不同的 class 文件当中，所以经常要从这个 class 文件中要调用另外一个 class 文件中的方法，如果另外一个文件不存在的话，则会引发系统异常。

而程序在启动的时候，并不会一次性加载程序所要用到的 class 文件，而是根据程序的需要，通过 Java 的类加载制（ClassLoader）来动态加载某个 class 文件到内存当的，从而只有 class 文件被载入到了内存之后，才能被其它 class 文件所引用。所以 ClassLoader 就是用来动态加载 class 件到内存当中用的。

双亲机制

类的加载就是虚拟机通过一个类的全限定名来获取描述此类的二进制字节流，而完成这个加载动作的就是类加载器。

类和类加载器息息相关，判定两个类是否相等，只有在这两个类被同一个类加载器加载的情况下才有意义，否则即便是两个类来自同一个 Class 文件，被不同类加载器加载，它们也是不相等的。

注：这里的相等性保函 Class 对象的 equals()方法、isAssignableFrom()方法、isInstance()方法的返回结果以及 Instance 关键字对对象所属关系的判定结果等。

类加载器可以分为三类：

启动类加载器（Bootstrap ClassLoader）：负责加载<JAVA_HOME>\lib

目录下或者被-Xbootclasspath 参数所指定的路径的，并且是被虚拟机所识别的库到内存中。

扩展类加载器（Extension ClassLoader）：负责加载<JAVA_HOME>\lib\ext

目录下或者被 java.ext.dirs 系统变量所指定的路径的所有类库到内存中。

应用类加载器（Application ClassLoader）：负责加载用户类路径上的指定类库，如果应用程序中没有实现自己的类加载器，一般就是这个类加载器去加载应用程序中的类库。

1、原理介绍

ClassLoader 使用的是双亲委托模型来搜索类的，每个 ClassLoader 实例都有一个父类加载器的引用（不是继承的关系，是一个包含的关系），虚拟机内置的类加载器（Bootstrap ClassLoader）本身没有父类加载器，但可以用作其它

lassLoader 实例的的父类加载器。

当一个 ClassLoader 实例需要加载某个类时，它会在试图搜索某个类之前，先把这个任务委托给它的父类加载器，这个过程是由上至下依次检查的，首先由最顶层的类加载器 Bootstrap ClassLoader 试图加载，如果没加载到，则把任务转交

给 Extension ClassLoader 试图加载，如果也没加载到，则转交给 App ClassLoader 进行加载，如果它也没有加载得到的话，则返回给委托的发起者，由它到指定的文件系统或网络等待 URL 中加载该类。

如果它们都没有加载到这个类时，则抛出 ClassNotFoundException 异常。否则将这个找到的类生成一个类的定义，将它加载到内存当中，最后返回这个类在内存中的 Class 实例对象。

类加载机制：

类的加载指的是将类的.class 文件中的二进制数据读入到内存中，将其放在运行时数据区的方法去内，然后在堆区创建一个 java.lang.Class 对象，用来封装在方法区内的数据结构。类的加载最终是在堆区内的 Class 对象，Class 对象封装了类在方法区内的数据结构，并且向 Java 程序员提供了访问方法区内的数据结构的接口。

类加载有三种方式：

1）命令行启动应用时候由 JVM 初始化加载

2）通过 Class.forName（）方法动态加载

3）通过 ClassLoader.loadClass（）方法动态加载

这么多类加载器，那么当类在加载的时候会使用哪个加载器呢？

这个时候就要提到类加载器的双亲委派模型，流程图如下所示：

双亲委派模型的整个工作流程非常的简单，如下所示：

如果一个类加载器收到了加载类的请求，它不会自己立去加载类，它会先去请求父类加载器，每个层次的类加器都是如此。层层传递，直到传递到最高层的类加载器只有当 父类加载器反馈自己无法加载这个类，才会有当子类加载器去加载该类。

2、为什么要使用双亲委托这种模型呢？

因为这样可以避免重复加载，当父亲已经加载了该类的时候，就没有必要让子 ClassLoader 再加载一次。

考虑到安全因素，我们试想一下，如果不使用这种委托模式，那我们就可以随时使用自定义的 String 来动态替代 java 核心 api 中定义的类型，这样会存在非常大的安全隐患，而双亲委托的方式，就可以避免这种情况，因为 String 已经在

启动时就被引导类加载器（BootstrcpClassLoader）加载，所以用户自定义的 ClassLoader 永远也无法加载一个自己写的 String，除非你改变 JDK 中 ClassLoader 搜索类的默认算法。

3、但是 JVM 在搜索类的时候，又是如何判定两个 class 是相同的呢？

JVM 在判定两个 class 是否相同时，不仅要判断两个类名否相同，而且要判断是否由同一个类加载器实例加载的。

只有两者同时满足的情况下，JVM 才认为这两个 class 是相同的。就算两个 class 是同一份 class 字节码，如果被两个不同的 ClassLoader 实例所加载，JVM 也会认为它们是两个不同 class。

比如网络上的一个 Java 类 org.classloader.simple.NetClassLoaderSimple，javac 编译之后生成字节码文件 NetClasLoaderSimple.class，ClassLoaderA 和 ClassLoaderB 这个类加载器并读取了 NetClassLoaderSimple.class 文件并分别定义出了 java.lang.Class 实例来表示这个类，对 JVM 来说，它们是两个不同的实例对象，但它们确实是一份字节码文件，如果试图将这个 Class 实例生成具体的对象进行转换时，就会抛运行时异常 java.lang.ClassCastException，提示这是两个不同的类型。

Android 类加载器

对于 Android 而言，最终的 apk 文件包含的是 dex 类型的文件，dex 文件是将 class 文件重新打包，打包的规则又不是简单地压缩，而是完全对 class 文件内部的各种函数表进行优化，产生一个新的文件，即 dex 文件。因此加载某种特殊的 Class 文件就需要特殊的类加载器 DexClassLoader。

可以动态加载 Jar 通过 URLClassLoader

1.ClassLoader 隔离问题：JVM 识别一个类是由 ClassLoaderid + PackageName + ClassName。

2.加载不同 Jar 包中的公共类：

- 让父 ClassLoader 加载公共的 Jar，子 ClassLoade 加载包含公共 Jar 的 Jar，此时子 ClassLoader 在加载 Jar 的时候会先去父 ClassLoader 中找。(只适用 Java)
- 重写加载包含公共 Jar 的 Jar 的 ClassLoader，在 loClass 中找到已经加载过公共 Jar 的 ClassLoader，是把父 ClassLoader 替换掉。(只适用 Java)
- 在生成包含公共 Jar 的 Jar 时候把公共 Jar 去掉。

5、JVM 跟 Art、Dalvik 对比？

6、GC 收集器简介？以及它的内存划分怎么样的？

（1）简介：

Garbage-First（G1，垃圾优先）收集器是服务类型的收集器，目标是多处理器机器、大内存机器。它高度符合垃圾收集暂停时间的目标，同时实现高吞吐量。Oracle JDK 7 update 4 以及更新发布版完全支持 G1 垃圾收集器

（2）G1 的内存划分方式：

它是将堆内存被划分为多个大小相等的 heap 区,每个 heap 区都是逻辑上连续的一段内存(virtual memory). 其中一部分区域被当成老一代收集器相同的角色(eden, survivor, old), 但每个角色的区域个数都不是固定的。这在内存使用上提供了更多的灵活性

7、Java 的虚拟机 JVM 的两个内存：栈内存和堆内存的区别是什么？

Java 把内存划分成两种：一种是栈内存，一种是堆内存。两者的区别是：

1）栈内存：在函数中定义的一些基本类型的变量和对象的引用变量都在函数的栈内存中分配。 当在一段代码块定义一个变量时，Java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java 会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。

2）堆内存：堆内存用来存放由 new 创建的对象和数组。在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理。

8、JVM 调优的常见命令行工具有哪些？JVM 常见的调优参数有哪些？

（1）JVM 调优的常见命令工具包括：

1）jps 命令用于查询正在运行的 JVM 进程，

2）jstat 可以实时显示本地或远程 JVM 进程中类装载、内存、垃圾收集、JIT 编译等数据

3）jinfo 用于查询当前运行这的 JVM 属性和参数的值。

4）jmap 用于显示当前 Java 堆和永久代的详细信息

5）jhat 用于分析使用 jmap 生成的 dump 文件，是 JDK 自带的工具

6）jstack 用于生成当前 JVM 的所有线程快照，线程快照是虚拟机每一条线程正在执行的方法,目的是定位线程出现长时间停顿的原因。

（2）JVM 常见的调优参数包括：

-Xmx

指定 java 程序的最大堆内存，使用 java -Xmx5000M -version 判断当前系统能分配的最大堆内存

-Xms

指定最小堆内存，通常设置成跟最大堆内存一样，减少 GC

-Xmn

设置年轻代大小。整个堆大小=年轻代大小 + 年老代大小。所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

-Xss

指定线程的最大栈空间，此参数决定了 java 函数调用的深度，值越大调用深度越深，若值太小则容易出栈溢出错误(StackOverflowError)

-XX:PermSize

指定方法区(永久区)的初始值,默认是物理内存的 1/64， 在 Java8 永久区移除, 代之的是元数据区， 由-XX:MetaspaceSize 指定

-XX:MaxPermSize

指定方法区的最大值, 默认是物理内存的 1/4， 在 java8 中由 -XX:MaxMetaspaceSize 指定元数据区的大小

-XX:NewRatio=n

年老代与年轻代的比值，-XX:NewRatio=2, 表示年老代与年轻代的比值为 2:1

-XX:SurvivorRatio=n

Eden 区与 Survivor 区的大小比值，-XX:SurvivorRatio=8 表示 Eden 区与 Survivor 区的大小比值是 8:1:1，因为 Survivor 区有两个(from, to)

9、jstack,jmap,jutil 分别的意义？如何线上排查 JVM 的相关问题？

10、JVM 方法区存储内容 是否会动态扩展 是否会出现内存溢出 出现的原因有哪些。

11、如何解决同时存在的对象创建和对象回收问题？

12、JVM 中最大堆大小有没有限制？

13、JVM 方法区存储内容 是否会动态扩展 是否会出现内存溢出 出现的原因有哪些。

14、如何理解 Java 的虚函数表？

15、Java 运行时数据区域，导致内存溢出的原因。

16、对象创建、内存布局，访问定位等。

第四章 Android 面试题

第一节 Android 基础面试题 （★ ★ ★）

1、什么是 ANR 如何避免它？

答：在 Android 上，如果你的应用程序有一段时间响应不够灵敏，系统会向用户显示一个对话框，这个对话框称作应 用程序无响应（ANR：Application NotResponding）对话框。 用户可以选择让程序继续运行，但是，他们在使用你的 应用程序时，并不希望每次都要处理这个对话框。因此 ，在程序里对响应性能的设计很重要这样，这样系统就不会显 示 ANR 给用户。

不同的组件发生 ANR 的时间不一样，Activity 是 5 秒，BroadCastReceiver 是 10 秒，Service 是 20 秒（均为前台）。

如果开发机器上出现问题，我们可以通过查看/data/anr/traces.txt 即可，最新的 ANR 信息在最开始部分。

- 主线程被 IO 操作（从 4.0 之后网络 IO 不允许在主线程中）阻塞。
- 主线程中存在耗时的计算
- 主线程中错误的操作，比如 Thread.wait 或者 Thread.sleep 等 Android 系统会监控程序的响应状况，一旦出现下面两种情况，则弹出 ANR 对话框

- 应用在 5 秒内未响应用户的输入事件（如按键或者触摸）
- BroadcastReceiver 未在 10 秒内完成相关的处理
- Service 在特定的时间内无法处理完成 20 秒

修正：

1、使用 AsyncTask 处理耗时 IO 操作。

2、使用 Thread 或者 HandlerThread 时，调用 Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND)设置优先级，否则仍然会降低程序响应，因为默认 Thread 的优先级和主线程相同。

3、使用 Handler 处理工作线程结果，而不是使用 Thread.wait()或者 Thread.sleep()来阻塞主线程。

4、Activity 的 onCreate 和 onResume 回调中尽量避免耗时的代码。

BroadcastReceiver 中 onReceive 代码也要尽量减少耗时，建议使用 IntentService 处理。

解决方案：

将所有耗时操作，比如访问网络，Socket 通信，查询大 量 SQL 语句，复杂逻辑计算等都放在子线程中去，然 后通过 handler.sendMessage、runonUIThread、AsyncTask、RxJava 等方式更新 UI。无论如何都要确保用户界面的流畅 度。如果耗时操作需要让用户等待，那么可以在界面上显示度条。

深入回答

2、Activity 和 Fragment 生命周期有哪些？

3、横竖屏切换时候 Activity 的生命周期

不设置 Activity 的 android:configChanges 时，切屏会重新回调各个生命周期，切横屏时会执行一次，切竖屏时会执行两次。 设置 Activity 的 android:configChanges="orientation"时，切屏还是会调用各个生命周期，切换横竖屏只会执行一次 设置 Activity 的 android:configChanges="orientation |keyboardHidden"时，切屏不会重新调用各个生命周期，只会执行 onConfigurationChanged 方法

4、AsyncTask 的缺陷和问题，说说他的原理。

AsyncTask 是什么？

AsyncTask 是一种轻量级的异步任务类，它可以在线程池中执行后台任务，然后把执行的进度和最终结果传递给主线程并在主线程中更新 UI。

AsyncTask 是一个抽象的泛型类，它提供了 Params、Progress 和 Result 这三个泛型参数，其中 Params 表示参数的类型，Progress 表示后台任务的执行进度和类型，而 Result 则表示后台任务的返回结果的类型，如果 AsyncTask 不需要传递具体的参数，那么这三个泛型参数可以用 Void 来代替。

关于线程池：

AsyncTask 对应的线程池 ThreadPoolExecutor 都是进程范围内共享的，且都是 static 的，所以是 Asynctask 控制着进程范围内所有的子类实例。由于这个限制的存在，当使用默认线程池时，如果线程数超过线程池的最大容量，线程池就会爆掉(3.0 后默认串行执行，不会出现个问题)。针对这种情况，可以尝试自定义线程池，配合 Asynctask 使用。

关于默认线程池：

AsyncTask 里面线程池是一个核心线程数为 CPU + 1，最大线程数为 CPU * 2 + 1，工作队列长度为 128 的线程池，线程等待队列的最大等待数为 28，但是可以自定义线程池。线程池是由 AsyncTask 来处理的，线程池允许 tasks 并行运行，需要注意的是并发情况下数据的一致性问题，新数据可能会被老数据覆盖掉。所以希望 tasks 能够串行运行的话，使用 SERIAL_EXECUTOR。

AsyncTask 在不同的 SDK 版本中的区别：

调用 AsyncTask 的 execute 方法不能立即执行程序的原因及改善方案通过查阅官方文档发现，AsyncTask 首次引入时，异步任务是在一个独立的线程中顺序的执行，也就是说一次只执行一个任务，不能并行的执行，从 1.6 开始，AsyncTask 引入了线程池，支持同时执行 5 个异步任务，也就是说只能有 5 个线程运行，超过的线程只能等待，等待前的线程直到某个执行完了才被调度和运行。换句话说，如果进程中的 AsyncTask 实例个数超过 5 个，那么假如前 5 都运行很长时间的话，那么第 6 个只能等待机会了。这是 AsyncTask 的一个限制，而且对于 2.3 以前的版本无法解决。如果你的应用需要大量的后台线程去执行任务，那么只能放弃使用 AsyncTask，自己创建线程池来管理 Thread。不得不说，虽然 AsyncTask 较 Thread 使用起来方便，但是它最多只能同时运行 5 个线程，这也大大局限了它的作用，你必须要小心设计你的应用，错开使用 AsyncTask 时间，尽力做到分时，或者保证数量不会大于 5 个，否就会遇到上面提到的问题。可能是 Google 意识到了 AsynTask 的局限性了，从 Android 3.0 开始对 AsyncTask 的 API 做出了一些调整：每次只启动一个线程执行一个任务，完了之后再执行第二个任务，也就是相当于只有一个后台线程在执行所提交的任务。

一些问题：

1.生命周期

很多开发者会认为一个在 Activity 中创建的 AsyncTask 会随着 Activity 的销毁而销毁。然而事实并非如此。AsynTask 会一直执行，直到 doInBackground()方法执行完毕，然后，如果 cancel(boolean)被调用,那么 onCancelled(Result result)方法会被执行；否则，执行 onPostExecute(Result result)方法。如果我们的 Activity 销毁之前，没有取消 AsyncTask，这有可能让我们的应用崩溃(crash)。因为它想要处理的 view 已经不存在了。所以，我们是必须确保在销毁活动之前取消任务。总之，我们使用 AsyncTask 需要确保 AsyncTask 正确的取消。

2.内存泄漏

如果 AsyncTask 被声明为 Activity 的非静态内部类，那么 AsyncTask 会保留一个对 Activity 的引用。如果 Activity 已经被销毁，AsyncTask 的后台线程还在执行，它将继续在内存里保留这个引用，导致 Activity 无法被回收，引起内存泄漏。

3.结果丢失

屏幕旋转或 Activity 在后台被系统杀掉等情况会导致 Activity 的重新创建，之前运行的 AsyncTask 会持有一个之前 Activity 的引用，这个引用已经无效，这时调用 onPostExecute()再去更新界面将不再生效。

4.并行还是串行

在 Android1.6 之前的版本，AsyncTask 是串行的，在 1.6 之后的版本，采用线程池处理并行任务，但是从 Android 3.0 开始，为了避免 AsyncTask 所带来的并发错误，又采用一个线程来串行执行任务。可以使用 executeOnExecutor()方法来并行地执行任务。

AsyncTask 原理

- AsyncTask 中有两个线程池（SerialExecutor 和 THREAD_POOL_EXECUTOR）和一个 Handler（InternalHandler），其中线程池 SerialExecutor 用于任务的排队，而线程池 THREAD_POOL_EXECUTOR 用于真正地执行任务，InternalHandler 用于将执行环境从线程池切换到主线程。
- sHandler 是一个静态的 Handler 对象，为了能够将执行环境切换到主线程，这就要求 sHandler 这个对象必须在主线程创建。由于静态成员会在加载类的时候进行初始化，因此这就变相要求 AsyncTask 的类必须在主线程中加载，否则同一个进程中的 AsyncTask 都将无法正常工作。

5、onSaveInstanceState() 与 onRestoreIntanceState()

Activity 的 onSaveInstanceState() 和 onRestoreInstanceState()并不是生命周期方法，它们不同于 onCreate()、onPause()等生命周期方法，它们并不一定会被触发。当应用遇到意外情况（如：内存不足、用户直接按 Home 键）由系统销毁一个 Activity 时，onSaveInstanceState() 会被调用。但是当用户主动去销毁一个 Activity 时，例如在应用中按返回键，onSaveInstanceState()就不会被调用。因为在这种情况下，用户的行为决定了不需要保存 Activity 的状态。通常 onSaveInstanceState()只适合用于保存一些临时性的状态，而 onPause()适合用于数据的持久化保存。 在 activity 被杀掉之前调用保存每个实例的状态,以保证该状态可以在 onCreate(Bundle)或者 onRestoreInstanceState(Bundle) (传入的 Bundle 参数是由 onSaveInstanceState 封装好的)中恢复。这个方法在一个 activity 被杀死前调用，当该 activity 在将来某个时刻回来时可以恢复其先前状态。例如，

如果 activity B 启用后位于 activity A 的前端，在某个时刻 activity A 因为系统回收资源的问题要被杀掉，A 通过 onSaveInstanceState 将有机会保存其用户界面状态，使得将来用户返回到 activity A 时能通过 onCreate(Bundle)或者 onRestoreInstanceState(Bundle)恢复界面的状态

深入理解

6、android 中进程的优先级？

1. 前台进程：

即与用户正在交互的 Activity 或者 Activity 用到的 Service 等，如果系统内存不足时前台进程是最晚被杀死的

2. 可见进程：

可以是处于暂停状态(onPause)的 Activity 或者绑定在其上的 Service，即被用户可见，但由于失了焦点而不能与用户交互

3. 服务进程：

其中运行着使用 startService 方法启动的 Service，虽然不被用户可见，但是却是用户关心的，例如用户正在非音乐界面听的音乐或者正在非下载页面下载的文件等；当系统要空间运行，前两者进程才会被终止

4. 后台进程：

其中运行着执行 onStop 方法而停止的程序，但是却不是用户当前关心的，例如后台挂着的 QQ，这时的进程系统一旦没了有内存就首先被杀死

5. 空进程：

不包含任何应用程序的进程，这样的进程系统是一般不会让他存在的

7、Bunder 传递对象为什么需要序列化？Serialzable 和 Parcelable 的区别？

因为 bundle 传递数据时只支持基本数据类型，所以在传递对象时需要序列化转换成可存储或可传输的本质状态（字节流）。序列化后的对象可以在网络、IPC（比如启动另一个进程的 Activity、Service 和 Reciver）之间进行传输，也可以存储到本地。

Serializable（Java 自带）：

Serializable 是序列化的意思，表示将一个对象转换成存储或可传输的状态。序列化后的对象可以在网络上进传输，也可以存储到本地。

Parcelable（android 专用）：

除了 Serializable 之外，使用 Parcelable 也可以实现相同的效果，不过不同于将对象进行序列化，Parcelable 方式的实现原理是将一个完整的对象进行分解，而分解后的每一部分都是 Intent 所支持的数据类型，这也就实现传递对象的功能了。

区别总结如下图所示：

8、动画

- tween 补间动画。通过指定 View 的初末状态和变化方式，对 View 的内容完成一系列的图形变换来实现动画效果。 Alpha, Scale ,Translate, Rotate。
- frame 帧动画。AnimationDrawable 控制 animation-list.xml 布局
- PropertyAnimation 属性动画 3.0 引入，属性动画核心思想是对值的变化。

Property Animation 动画有两个步聚：

1.计算属性值

2.为目标对象的属性设置属性值，即应用和刷新动画

计算属性分为 3 个过程：

过程一：

计算已完成动画分数 elapsed fraction。为了执行一个动画，你需要创建一个 ValueAnimator，并且指定目标对象属性的开始、结束和持续时间。在调用 start 后的整个动画过程中，ValueAnimator 会根据已经完成的动画时间计算得到一个 0 到 1 之间的分数，代表该动画的已完成动画百分比。0 表示 0%，1 表示 100%。

过程二：

计算插值（动画变化率）interpolated fraction 。当 ValueAnimator 计算完已完成的动画分数后，它会调用当前设置的 TimeInterpolator，去计算得到一个 interpolated（插值）分数，在计算过程中，已完成动画百分比会被加入到新的插值计算中。

过程三：

计算属性值当插值分数计算完成后，ValueAnimator 会根据插值分数调用合适的 TypeEvaluator 去计算运动中的属性值。 以上分析引入了两个概念：已完成动画分数（elapsed fraction）、插值分数(interpolated fraction)。

原理及特点：

1.属性动画：

插值器：作用是根据时间流逝的百分比来计算属性变化的百分比

估值器：在 1 的基础上由这个东西来计算出属性到底变化了多少数值的类

其实就是利用插值器和估值器，来计出各个时刻View的属性，然后通过改变View的属性来实现 View 的动画效果。

2.View 动画:

只是影像变化，view 的实际位置还在原来地方。

3.帧动画：

是在 xml 中定义好一系列图片之后，使用 AnimatonDrawable 来播放的动画。

它们的区别：

属性动画才是真正的实现了 view 的移动，补间动画对 view 的移动更像是在不同地方绘制了一个影子，实际对象还是处于原来的地方。当动画的 repeatCount 设置为无限循环时，如果在 Activity 退出时没有及时将动画停止，属性动画会导致 Activity 无法释放而导致内存泄漏，而补间动画却没问题。 xml 文件实现的补间动画，复用率极高。在 Activity 切换，窗口弹出时等情景中有着很好的效果。使用帧动画时需要注意，不要使用过多特别大的图，容导致内存不足。

为什么属性动画移动后仍可点击？

播放补间动画的时候，我们所看到的变化，都只是临时的。而属性动画呢，它所改变的东西，却会更新到这个 View 所对应的矩阵中，所以当 ViewGroup 分派事件的时候，会正确的将当前触摸坐标，转换成矩阵变化后的坐标，这就是为什么播放补间动画不会改变触摸区域的原因了。

9、Context 相关

1、Activity 和 Service 以及 Application 的 Context 是不一样的,Activity 继承自 ContextThemeWraper.其他的继承自 ContextWrapper。

2、每一个 Activity 和 Service 以及 Application 的 Context 是一个新的 ContextImpl 对象。

3、getApplication()用来获取 Application 实例的，但是这个方法只有在 Activity 和 Service 中才能调用的到。那也许在绝大多数情况下我们都是在 Activity 或者 Servic 中使用 Application 的，但是如果在一些其它的场景，比如 BroadcastReceiver 中也想获得 Application 的实例，这时就可以借助 getApplicationContext()方法，getApplicationContext()比 getApplication()方法的作用域会更广一些，任何一个 Context 的实例，只要调用 getApplicationContext()方法都可以拿到我们的 Application 对象。

4、创建对话框时不可以用 Application 的 context，只能用 Activity 的 context。

5、Context 的数量等于 Activity 的个数 + Service 的个数 +1，这个 1 为 Application。

10、Android 各版本新特性

Android5.0 新特性

MaterialDesign 设计风格

支持 64 位 ART 虚拟机（5.0 推出的 ART 虚拟机，在 5.0 之前都是 Dalvik。他们的区别是： Dalvik,每次运行,字节码都需要通过即时编译器转换成机器码(JIT)。 ART,第一次安装应用的时候,字节码就会预先编译成机器码(AOT)）

通知详情可以用户自己设计

Android6.0 新特性

动态权限管理

支持快速充电的切换

支持文件夹拖拽应用

相机新增专业模式

Android7.0 新特性

多窗口支持

V2 签名

增强的 Java8 语言模式

夜间模式

Android8.0（O）新特性

优化通知

通知渠道 (Notification Channel) 通知标志 休眠 通知超时 通知设置 通知清除

画中画模式：清单中 Activity 设置 android:supportsPictureInPicture

后台限制

自动填充框架

系统优化

等等优化很多

Android9.0（P）新特性

室内 WIFI 定位

“刘海”屏幕支持

安全增强

等等优化很多

Android10.0（Q）目前曝光的新特性

- **夜间模式**：包括手机上的所有应用都可以为其设置暗黑模式。
- **桌面模式**：提供类似于 PC 的体验，但是远远不能代替 PC。
- **屏幕录制**：通过长按“电源”菜单中的"屏幕快照"来开启。

11、Json

JSON 的全称是 JavaScript Object Notation，也就是 JavaScript 对象表示法

JSON 是存储和交换文本信息的语法，类似 XML，但是比 XML 更小、更快，更易解析 JSON 是轻量级的文本数据交换格式，独立于语言，具有可描述性，更易理解，对象可以包含多个名称/值对，比如：

```
{"name":"zhangsan" , "age":25}
```

使用谷歌的 GSON 包进行解析，在 Android Studio 里引入依赖：

```
compile 'com.google.code.gson:gson:2.7'
```

值得注意的是实体类中变量名称必须和 json 中的值名字相同。

使用示例：

1、解析成实体类：

```
Gson gson = new Gson();

Student student = gson.fromJson(json1, Student.class);
```

2、解析成 int 数组：

```
Gson gson = new Gson();

int[] ages = gson.fromJson(json2, int[].class);
```

3、直接解析成 List.

```
Gson gson = new Gson();

List<Integer> ages = gson.fromJson(json2,
newTypeToken<List<Integer>>(){}.getType);



Gson gson = new Gson();

List<Student> students = gson.fromJson(json3,
newTypeToke<List<Student>>(){}.getType);
```

优点：

- 轻量级的数据交换格式
- 读写更加容易
- 易于机器的解析和生成

缺点：

- 语义性较差，不如 xml 直观

12、android 中有哪几种解析 xml 的类,官方推荐哪种？以及它们的原理和区别？

DOM 解析

优点:

1.XML 树在内存中完整存储,因此可以直接修改其数据结构.

2.可以通过该解析器随时访问 XML 树中的任何一个节点.

3.DOM 解析器的 API 在使用上也相对比较简单.

缺点:

如果 XML 文档体积比较大时,将文档读入内存是非消耗系统资源的.

使用场景:

- DOM 是与平台和语言无关的方式表示 XML 文档的官方 W3C 标准.
- DOM 是以层次结构组织的节点的集合.这个层次结构允许开人员在树中寻找特定信息.分析该结构通常需要加载整个文档和构造层次结构,然后才能进行任何工作.
- DOM 是基于对象层次结构的.

SAX 解析

优点:

SAX 对内存的要求比较低,因为它让开发人员自己来决定所要处理的标签.特别是当开发人员只需要处理文档中包含的部分数据时,SAX 这种扩展能力得到了更好的体现.

缺点:

用 SAX 方式进行 XML 解析时,需要顺序执行,所以很难访问同一文档中的不同数据.此外,在基于该方式的解析编码程序也相对复杂.

使用场景:

对于含有数据量十分巨大,而又不用对文档的所有数据行遍历或者分析的时候,使用该方法十分有效.该方法不将整个文档读入内存,而只需读取到程序所需的文档标记处即可.

Xmlpull 解析

android SDK 提供了 xmlpullapi,xmlpull 和 sax 类似,是基于流（stream）操作文件,后者根据节点事件回调开发者编写的处理程序.因为是基于流的处理,因此 xmlpull 和 sax 都比较节约内存资源,不会像 dom 那样要把所有节点以对象树的形式展现在内存中.xmpull 比 sax 更简明,而且不需要扫描完整个流.

13、Jar 和 Aar 的区别

Jar 包里面只有代码，aar 里面不光有代码还包括资源文件，比如 drawable 文件，xml 资源文件。对于一些不常变动的 Android Library，我们可以直接引用 aar，加快编译速度。

14、Android 为每个应用程序分配的内存大小是多少

android 程序内存一般限制在 16M，也有的是 24M。近几年手机发展较快，一般都会分配两百兆左右，和具体机型有关。

15、更新 UI 方式

- Activity.runOnUiThread(Runnable)

- View.post(Runnable)，View.postDelay(Runnable, long)（可以理解为在当前操作视图 UI 线程添加队列）
- Handler
- AsyncTask
- Rxjava
- LiveData

16、ContentProvider 使用方法。

进行跨进程通信，实现进程间的数据交互和共享。通过 Context 中 getContentResolver() 获得实例，通过 Uri 匹配进行数据的增删改查。ContentProvider 使用表的形式来组织数据，无论数据的来源是什么，ConentProvider 都会认为是一种表，然后把数据组织成表格。

17、Thread、AsyncTask、IntentService 的使用场景与特点。

Thread 线程，独立运行与于 Activity 的，当 Activity 被 finish 后，如果没有主动停止 Thread 或者 run 方法没有执行完，其会一直执行下去。

AsyncTask 封装了两个线程池和一个 Handler（SerialExecutor 用于排队，THREAD_POOL_EXECUTOR 为真正的执行任务，Handler 将工作线程切换到主线程），其必须在 UI 线程中创建，execute 方法必须在 UI 线程中执行，一个任务实例只允许执行一次，执行多次抛出异常，用于网络请求或者简单数据处理。

IntentService：处理异步请求，实现多线程，在 onHandleIntent 中处理耗时操作，多个耗时任务会依次执行，执行完毕自动结束。

18、Merge、ViewStub 的作用。

Merge: 减少视图层级，可以删除多余的层级。

ViewStub: 按需加载，减少内存使用量、加快渲染速度、不支持 merge 标签。

19、activity 的 startActivity 和 context 的 startActivity 区别？

(1)、从 Activity 中启动新的 Activity 时可以直接 mContext.startActivity(intent)就好

(2)、如果从其他 Context 中启动 Activity 则必须给 intent 设置 Flag:

```
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK) ;

mContext.startActivity(intent);
```

20、怎么在 Service 中创建 Dialog 对话框？

1.在我们取得 Dialog 对象后，需给它设置类型，即：

```
dialog.getWindow().setType(WindowManager.LayoutParams.TYPE_SYSTEM_ALERT)
```

2.在 Manifest 中加上权限:

```
<uses-permission android:name="android.permission.SYSTEM_ALERT_WINOW" />
```

21、Asset 目录与 res 目录的区别？

assets：不会在 R 文件中生成相应标记，存放到这里的资源在打包时会打包到程序安装包中。（通过 AssetManager 类访问这些文件）

res：会在 R 文件中生成 id 标记，资源在打包时如果使用到则打包到安装包中，未用到不会打入安装包中。

res/anim：存放动画资源。

res/raw：和 asset 下文件一样，打包时直接打入程序安装包中（会映射到 R 文件中）。

22、Android 怎么加速启动 Activity?

- onCreate() 中不执行耗时操作 把页面显示的 View 细分一下，放在 AsyncTask 里逐步显示，用 Handler 更好。这样用户的看到的就是有层次有步骤的一个个的 View 的展示，不会是先看到一个黑屏，然后一下显示所有 View。最好做成动画，效果更自然。
- 利用多线程的目的就是尽可能的减少 onCreate() 和 onReume() 的时间，使得用户能尽快看到页面，操作页面。
- 减少主线程阻塞时间。
- 提高 Adapter 和 AdapterView 的效率。
- 优化布局文件。

23、Handler 机制

Android 消息循环流程图如下所示：

主要涉及的角色如下所示：

- message：消息。
- MessageQueue：消息队列，负责消息的存储与管理，负责管理由 Handler 发送过来的 Message。读取会自动删除消息，单链表维护，插入和删除上有优势。在其 next()方法中会无限循环，不断判断是否有消息，有就返回这条消息并移除。
- Looper：消息循环器，负责关联线程以及消息的分发，在该线程下从 MessageQueue 获取 Message，分发给 Handler，Looper 创建的时候会

创建一个 MessageQueue，调用 loop()方法的时候消息循环开始，其中会不断调用 messageQueue 的 next()方法，当有消息就处理，否则阻塞在 messageQueue 的 next()方法中。当 Looper 的 quit()被调用的时候会调用 messageQueue 的 quit()，此时 next()会返回 null，然后 loop()方法也就跟着退出。

- Handler：消息处理器，负责发送并处理消息，面向开发者，提供 API，并隐藏背后实现的细节。

整个消息的循环流程还是比较清晰的，具体说来：

- 1、Handler 通过 sendMessage()发送消息 Message 到消息队列 MessageQueue。
- 2、Looper 通过 loop()不断提取触发条件的 Message，并将 Message 交给对应的 target handler 来处理。
- 3、target handler 调用自身的 handleMessage()方法来处理 Message。

事实上，在整个消息循环的流程中，并不只有 Java 层参与，很多重要的工作都是在 C++层来完成的。我们来看下这些类的调用关系。

注：虚线表示关联关系，实线表示调用关系。

在这些类中 MessageQueue 是 Java 层与 C++层维系的桥梁，MessageQueue 与 Looper 相关功能都通过 MessageQueue 的 Native 方法来完成，而其他虚线连接的类只有关联关系，并没有直接调用的关系，它们发生关联的桥梁是 MessageQueue。

总结

- Handler 发送的消息由 MessageQueue 存储管理，并由 Looper 负责回调消息到 handleMessage()。
- 线程的转换由 Looper 完成，handleMessage() 所在线程由 Looper.loop() 调用者所在线程决定。

Handler 引起的内存泄露原因以及最佳解决方案

Handler 允许我们发送延时消息，如果在延时期间用户关闭了 Activity，那么该 Activity 会泄露。 这个泄露是因为 Message 会持有 Handler，而又因为 Java 的特性，内部类会持有外部类，使得 Activity 会被 Handler 持有，这样最终就导致 Activity 泄露。

解决：将 Handler 定义成静态的内部类，在内部持有 Activity 的弱引用，并在 Acitivity 的 onDestroy()中调用 handler.removeCallbacksAndMessages(null)及时移除所有消息。

为什么我们能在主线程直接使用 Handler，而不需要创建 Looper ？

通常我们认为 ActivityThread 就是主线程。事实上它并不是一个线程，而是主线程操作的管理者。在 ActivityThread.main() 方法中调用了 Looper.prepareMainLooper() 方法创建了 主线程的 Looper ,并且调用了 loop() 方法，所以我们就可以直接使用 Handler 了。

因此我们可以利用 Callback 这个拦截机制来拦截 Handler 的消息。如大部分插件化框架中 Hook ActivityThread.mH 的处理。

主线程的 Looper 不允许退出

主线程不允许退出，退出就意味 APP 要挂。

Handler 里藏着的 Callback 能干什么？

Handler.Callback 有优先处理消息的权利，当一条消息被 Callback 处理并拦截（返回 true），那么 Handler 的 handleMessage(msg) 方法就不会被调用了；如果 Callback 处理了消息，但是并没有拦截，那么就意味着一个消息可以同时被 Callback 以及 Handler 处理。

创建 Message 实例的最佳方式

为了节省开销，Android 给 Message 设计了回收机制，所以我们在使用的时候尽量复用 Message ，减少内存消耗：

- 通过 Message 的静态方法 Message.obtain()；
- 通过 Handler 的公有方法 handler.obtainMessage()。

子线程里弹 Toast 的正确姿势

本质上是因为 Toast 的实现依赖于 Handler，按子线程使用 Handler 的要求修改即可，同理的还有 Dialog。

妙用 Looper 机制

- 将 Runnable post 到主线程执行；
- 利用 Looper 判断当前线程是否是主线程。

主线程的死循环一直运行是不是特别消耗 CPU 资源呢？

并不是，这里就涉及到 Linux pipe/epoll 机制，简单说就是在主线程的 MessageQueue 没有消息时，便阻塞在 loop 的 queue.next()中的 nativePollOnce()方法里，此时主线程会释放 CPU 资源进入休眠状态，直到下个消息到达或者有事务发生，通过往 pipe 管道写端写入数据来唤醒主线程工作。这里采用的 epoll 机制，是一种 IO 多路复用机制，可以同时监控多个描述符，当某个描述符就绪(读或写就绪)，则立刻通知相应程序进行读或写操作，本质是同步 I/O，即读写是阻塞的。所以说，主线程大多数时候都是处于休眠状态，并不会消耗大量 CPU 资源。

handler postDelay 这个延迟是怎么实现的？

handler.postDelay 并不是先等待一定的时间再放入到 MessageQueue 中，而是直接进入 MessageQueue，以 MessageQueue 的时间顺序排列和唤醒的方式结合实现的。

如何保证在 msg.postDelay 情况下保证消息次序？

Handler 都没搞懂，拿什么去跳槽啊？

24、程序 A 能否接收到程序 B 的广播？

能，使用全局的 BroadCastRecevier 能进行跨进程通信，但是注意它只能被动接收广播。此外，LocalBroadCastRecevier 只限于本进程的广播间通信。

25、数据加载更多涉及到分页，你是怎么实现的？

分页加载就是一页一页加载数据，当滑动到底部、没有更多数据加载的时候，我们可以手动调用接口，重新刷新 RecyclerView。

26、通过 google 提供的 Gson 解析 json 时，定义 JavaBean 的规则是什么？

1). 实现序列化 Serializable

2). 属性私有化，并提供 get，set 方法

3). 提供无参构造

4). 属性名必须与 json 串中属性名保持一致 （因为 Gson 解析 json 串底层用到了 Java 的反射原理）

27、json 解析方式的两种区别？

1，SDK 提供 JSONArray，JSONObject

2，google 提供的 Gson 通过 fromJson()实现对象的反序列化（即将 json 串转换为对象类型） 通过 toJson()实现对象的序列化 （即将对象类型转换为 json 串）

28、线程池的相关知识。

Android 中的线程池都是直接或间接通过配置 ThreadPoolExecutor 来实现不同特性的线程池.Android 中最常见的类具有不同特性的线程池分别为 FixThreadPool、CachedhreadPool、SingleThreadPool、ScheduleThreadExecutr.

1).FixThreadPool

只有核心线程,并且数量固定的,也不会被回收,所有线程都活动时,因为队列没有限制大小,新任务会等待执行.

优点:更快的响应外界请求.

2).SingleThreadPool

只有一个核心线程,确保所有的任务都在同一线程中按序完成.因此不需要处理线程同步的问题.

3).CachedThreadPool

只有非核心线程,最大线程数非常大,所有线程都活动时会为新任务创建新线程,否则会利用空闲线程(60s 空闲时间,过了就会被回收,所以线程池中有 0 个线程的可能)处理任务.

优点:任何任务都会被立即执行(任务队列 SynchronousQuue 相当于一个空集合);比较适合执行大量的耗时较少的任务.

4).ScheduledThreadPool

核心线程数固定,非核心线程（闲着没活干会被立即回收数）没有限制.

优点:执行定时任务以及有固定周期的重复任务

29、内存泄露，怎样查找，怎么产生的内存泄露？

1.资源对象没关闭造成的内存泄漏

描述： 资源性对象比如(Cursor，File 文件等)往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于 java 虚拟机内，还存在于 java 虚拟机外。如果我们仅仅是把它的引用设置为 null,而不关闭它们，往往会造成内存泄漏。因为有些资源性对象，比如 SQLiteCursor(在析构函数 finalize(),如果我们没有关闭它，它自己会调 close()关闭)，如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该调用它的 close()函数，将其关闭

掉，然后才置为 null.在我们的程序退出时一定要确保我们的资源性对象已经关闭。

程序中经常会进行查询数据库的操作，但是经常会有使用完毕 Cursor 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

2.构造 Adapter 时，没有使用缓存的 convertView

描述： 以构造 ListView 的 BaseAdapter 为例，在 BaseAdapter 中提供了方法：public View getView(int position, ViewconvertView, ViewGroup parent) 来向 ListView 提供每一个 item 所需要的 view 对象。初始时 ListView 会从 BaseAdapter 中根据当前的屏幕布局实例化一定数量的 view 对象，同时 ListView 会将这些 view 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 list item 的 view 对象会被回收，然后被用来构造新出现的最下面的 list item。这个构造过程就是由 getView()方法完成的，getView()的第二个形参 View convertView 就是被缓存起来的 list item 的 view 对象(初始化时缓存中没有 view 对象则 convertView 是 null)。由此可以看出，如果我们不去使用 convertView，而是每次都在 getView() 中重新实例化一个 View 对象的话，即浪费资源也浪费时间，也会使得内存占用越来越大。 ListView 回收 list item 的 view 对象的过程可以查看: android.widget.AbsListView.java --> voidaddScrapView(View scrap) 方法。 示例代码：

```
public View getView(int position, ViewconvertView, ViewGroup parent) {

View view = new Xxx(...);

... ...

return view;
```

```
}
```

修正示例代码：

```
public View getView(int position, ViewconvertView, ViewGroup parent) {

View view = null;

if (convertView != null) {

view = convertView;

populate(view, getItem(position));

...

} else {

view = new Xxx(...);

...

}

return view;

}
```

3.Bitmap 对象不在使用时调用 recycle()释放内存

描述： 有时我们会手工的操作 Bitmap 对象，如果一个 Bitmap 对象比较占内存，当它不在被使用的时候，可以调用 Bitmap.recycle()方法回收此对象的像素所占用的内存，但这不是必须的，视情况而定。可以看一下代码中的注释：

/* •Free up the memory associated with thisbitmap's pixels, and mark the •bitmap as "dead", meaning itwill throw an exception if getPixels() or •setPixels() is called, and will drawnothing. This operation cannot be •reversed, so it should only be called ifyou are sure there are no •further uses for the bitmap. This is anadvanced call, and normally need •not be called, since the

normal GCprocess will free up this memory when •there are no more references to thisbitmap. /

4.试着使用关于 application 的 context 来替代和 activity 相关的 context

这是一个很隐晦的内存泄漏的情况。有一种简单的方法来避免 context 相关的内存泄漏。最显著地一个是避免 context 逃出他自己的范围之外。使用 Application context。这个 context 的生存周期和你的应用的生存周期一样长，而不是取决于 activity 的生存周期。如果你想保持一个长期生存的对象，并且这个对象需要一个 context,记得使用 application 对象。你可以通过调用 Context.getApplicationContext() or Activity.getApplication()来获得。更多的请看这篇文章如何避免 Android 内存泄漏。

5.注册没取消造成的内存泄漏

一些 Android 程序可能引用我们的 Anroid 程序的对象(比如注册机制)。即使我们的 Android 程序已经结束了，但是别的引用程序仍然还有对我们的 Android 程序的某个对象的引用，泄漏的内存依然不能被垃圾回收。调用 registerReceiver 后未调用 unregisterReceiver。 比如:假设我们希望在锁屏界面(LockScreen)中，监听系统中的电话服务以获取一些信息(如信号强度等)，则可以在 LockScreen 中定义一个 PhoneStateListener 的对象，同时将它注册到 TelephonyManager 服务中。对于 LockScreen 对象，当需要显示锁屏界面的时候就会创建一个 LockScreen 对象，而当锁屏界面消失的时候 LockScreen 对象就会被释放掉。 但是如果在释放 LockScreen 对象的时候忘记取消我们之前注册的 PhoneStateListener 对象，则会导致 LockScreen 无法被垃圾回收。如果不断的

使锁屏界面显示和消失，则最终会由于大量的 LockScreen 对象没有办法被回收而引起 OutOfMemory,使得 system_process 进程挂掉。 虽然有些系统程序，它本身好像是可以自动取消注册的(当然不及时)，但是我们还是应该在我们的程序中明确的取消注册，程序结束时应该把所有的注册都取消掉。

6.集合中对象没清理造成的内存泄漏

我们通常把一些对象的引用加入到了集合中，当我们不需要该对象时，并没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就更严重了。

查找内存泄漏可以使用 Android Studio 自带的 AndroidProfiler 工具或 MAT，也可以使用 Square 产品的 LeakCanary.

1、使用 AndroidProfiler 的 MEMORY 工具：

运行程序，对每一个页面进行内存分析检查。首先，反复打开关闭页面 5 次，然后收到 GC（点击 Profile MEMORY 左上角的垃圾桶图标），如果此时 total 内存还没有恢复到之前的数值，则可能发生了内存泄露。此时，再点击 Profile MEMORY 左上角的垃圾桶图标旁的 heap dump 按钮查看当前的内存堆栈情况，选择按包名查找，找到当前测试的 Activity，如果引用了多个实例，则表明发生了内存泄露。

2、使用 MAT：

1、运行程序，所有功能跑一遍，确保没有改出问题，完全退出程序，手动触发 GC，然后使用 adb shell dumpsys meminfo packagename -d 命令查看退出界面后 Objects 下的 Views 和 Activities 数目是否为 0，如果不是则通过 Leakcanary 检查可能存在内存泄露的地方，最后通过 MAT 分析，如此反复，改善满意为止。

1、在使用 MAT 之前，先使用 as 的 Profile 中的 Memory 去获取要分析的堆内存快照文件.hprof，如果要测试某个页面是否产生内存泄漏，可以先 dump 出没进入该页面的内存快照文件.hprof，然后，通常执行 5 次进入/退出该页面，然后再 dump 出此刻的内存快照文件.hprof，最后，将两者比较，如果内存相除明显，则可能发生内存泄露。（注意:MAT 需要标准的.hprof 文件，因此在 as 的 Profiler 中 GC 后 dump 出的内存快照文件.hprof 必须手动使用 android sdk platform-tools 下的 hprof-conv 程序进行转换才能被 MAT 打开）

2、然后，使用 MAT 打开前面保存的 2 份.hprof 文件，打开 Overview 界面，在 Overview 界面下面有 4 中 action，其中最常用的就是 Histogram 和 Dominator Tree。

Dominator Tree：支配树，按对象大小降序列出对象和其所引用的对象，注重引用关系分析。选择 Group by package，找到当前要检测的类（或者使用顶部的 Regex 直接搜索），查看它的 Object 数目是否正确，如果多了，则判断发生了内存泄露。然后，右击该类，选择 Merge Shortest Paths to GC Root 中的 exclude all phantom/weak/soft etc.references 选项来查看该类的 GC 强引用链。最后，通过引用链即可看到最终强引用该类的对象。

Histogram：直方图注重量的分析。使用方式与 Dominator Tree 类似。

3、对比 hprof 文件，检测出复杂情况下的内存泄露：

通用对比方式：在 Navigation History 下面选择想要对比的 dominator_tree/histogram，右击选择 Add to Compare Basket，然后在 Compare Basket 一栏中点击红色感叹号（Compare the results）生成对比表格（Compared

Tables），在顶部 Regex 输入要检测的类，查看引用关系或对象数量去进行分析即可。

针对于 Historam 的快速对比方式：直接选择 Histogram 上方的 Compare to another Heap Dump 选择要比较的 hprof 文件的 Historam 即可。

30、类的初始化顺序依次是？

（静态变量、静态代码块）>（变量、代码块）>构造方法

31、JSON 的结构？

json 是一种轻量级的数据交换格式， json 简单说就是对象和数组，所以这两种结构就是对象和数组两种结构，通过这两种结构可以表示各种复杂的结构

1、对象：对象表示为“{}”扩起来的内容，数据结构为 {key：value,key：value,...} 的键值对的结构，在面向对象的语言中，key 为对象的属性，value 为对应的属性值，所以很容易理解，取值方法为 对象.key 获取属性值，这个属性值的类型可以是 数字、字符串、数组、对象几种。

2、数组：数组在 json 中是中括号“[]”扩起来的内容，数据结构为 ["java","javascript","vb",...]，取值方式和所有语言中一样，使用索引获取，字段值的类型可以是 数字、字符串、数组、对象几种。 经过对象、数组 2 种结构就可以组合成复杂的数据结构了。

32、ViewPager 使用细节，如何设置成每次只初始化当前的 Fragment，其他的不初始化（提示：Fragment 懒加载）？

自定义一个 LazyLoadFragment 基类，利用 setUserVisibleHint 和 生命周期方法，通过对 Fragment 状态判断，进行数据加载，并将数据加载的接口提供开放出去，供子类使用。然后在子类 Fragment 中实现 requestData 方法即可。

这里添加了一个 isDataLoaded 变量，目的是避免重复加载数据。考虑到有时候需要刷新数据的问题，便提供了一个用于强制刷新的参数判断。

```
public abstract class LazyLoadFragment extends BaseFragment {

    protected boolean isViewInitiated;

    protected boolean isDataLoaded;

    @Override

    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

    }

    @Override

    public void onActivityCreated(Bundle savedInstanceState) {

        super.onActivityCreated(savedInstanceState);

        isViewInitiated = true;

        prepareRequestData();

    }

    @Override

    public void setUserVisibleHint(boolean isVisibleToUser) {

        super.setUserVisibleHint(isVisibleToUser);

        prepareRequestData();

    }

    public abstract void requestData();

    public boolean prepareRequestData() {

        return prepareRequestData(false);

    }

    public boolean prepareRequestData(boolean forceUpdate) {
```

```
        if (getUserVisibleHint() && isViewInitiated && (!isDataLoaded ||
forceUpdate)) {

            requestData();

            isDataLoaded = true;

            return true;

        }

        return false;

    }

}
```

35、Android 为什么引入 Parcelable？

可以肯定的是，两者都是支持序列化和反序列化的操作。

两者最大的区别在于 存储媒介的不同，Serializable 使用 I/O 读写存储在硬盘上，而 Parcelable 是直接 在内存中读写。很明显，内存的读写速度通常大于 IO 读写，所以在 Android 中传递数据优先选择 Parcelable。

Serializable 会使用反射，序列化和反序列化过程需要大量 I/O 操作，Parcelable 自己实现封送和解封（marshalled &unmarshalled）操作不需要用反射，数据也存放在 Native 内存中，效率要快很多。

36、有没有尝试简化 Parcelable 的使用？

使用 Parcelable 插件（Android Parcelable code generator）进行实体类的序列化的实现。

37、Bitmap 使用时候注意什么？

1、要选择合适的图片规格（bitmap 类型）：

```
ALPHA_8    每个像素占用 1byte 内存
```

```
ARGB_4444 每个像素占用 2byte 内存

ARGB_8888 每个像素占用 4byte 内存（默认）

RGB_565 每个像素占用 2byte 内存
```

2、降低采样率。BitmapFactory.Options 参数 inSampleSize 的使用，先把 options.inJustDecodeBounds 设为 true，只是去读取图片的大小，在拿到图片的大小之后和要显示的大小做比较通过 calculateInSampleSize()函数计算 inSampleSize 的具体值，得到值之后。options.inJustDecodeBounds 设为 false 读图片资源。

3、复用内存。即，通过软引用(内存不够的时候才会回收掉)，复用内存块，不需要再重新给这个 bitmap 申请一块新的内存，避免了一次内存的分配和回收，从而改善了运行效率。

4、使用 recycle()方法及时回收内存。

5、压缩图片。

38、Oom 是否可以 try catch ？

只有在一种情况下，这样做是可行的：

在 try 语句中声明了很大的对象，导致 OOM，并且可以确认 OOM 是由 try 语句中的对象声明导致的，那么在 catch 语句中，可以释放掉这些对象，解决 OOM 的问题，继续执行剩余语句。

但是这通常不是合适的做法。

Java 中管理内存除了显式地 catch OOM 之外还有更多有效的方法：比如 SoftReference, WeakReference, 硬盘缓存等。 在 JVM 用光内存之前，会多次触

发 GC，这些 GC 会降低程序运行的效率。 如果 OOM 的原因不是 try 语句中的对象（比如内存泄漏），那么在 catch 语句中会继续抛出 OOM。

39、多进程场景遇见过么？

1、在新的进程中，启动前台 Service，播放音乐。 2、一个成熟的应用一定是多模块化的。首先多进程开发能为应用解决了 OOM 问题，因为 Android 对内存的限制是针对于进程的，所以，当我们需要加载大图之类的操作，可以在新的进程中去执行，避免主进程 OOM。而且假如图片浏览进程打开了一个过大的图片，java heap 申请内存失败，该进程崩溃并不影响我主进程的使用。

40、Canvas.save()跟 Canvas.restore()的调用时机

save：用来保存 Canvas 的状态。save 之后，可以调用 Canvas 的平移、放缩、旋转、错切、裁剪等操作。

restore：用来恢复 Canvas 之前保存的状态。防止 save 后对 Canvas 执行的操作对后续的绘制有影响。

save 和 restore 要配对使用（restore 可以比 save 少，但不能多），如果 restore 调用次数比 save 多，会引发 Error。save 和 restore 操作执行的时机不同，就能造成绘制的图形不同。

41、数据库升级增加表和删除表都不涉及数据迁移，但是修改表涉及到对原有数据进行迁移。升级的方法如下所示：

将现有表命名为临时表。 创建新表。 将临时表的数据导入新表。 删除临时表。

如果是跨版本数据库升级，可以有两种方式，如下所示：

逐级升级，确定相邻版本与现在版本的差别，V1 升级到 V2,V2 升级到 V3，依次类推。 跨级升级，确定每个版本与现在数据库的差别，为每个 case 编写专门升级大代码。

```
public class DBservice extends SQLiteOpenHelper{

    private String CREATE_BOOK = "create table book(bookId integer
primarykey,bookName text);";

    private String CREATE_TEMP_BOOK = "alter table book rename to _temp_book";

    private String INSERT_DATA = "insert into book select *,'' from _temp_book";

    private String DROP_BOOK = "drop table _temp_book";

    public DBservice(Context context, String name, CursorFactory factory,int
version) {

    super(context, name, factory, version);

    }

    @Override

    public void onCreate(SQLiteDatabase db) {

        db.execSQL(CREATE_BOOK);

    }

    @Override

    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
{

    switch (newVersion) {

        case 2:

            db.beginTransaction();



            db.execSQL(CREATE_TEMP_BOOK);

            db.execSQL(CREATE_BOOK);

            db.execSQL(INSERT_DATA);

            db.execSQL(DROP_BOOK);



            db.setTransactionSuccessful();
```

```
            db.endTransaction();



            break;

    }

}
```

42、编译期注解跟运行时注解

运行期注解(RunTime)利用反射去获取信息还是比较损耗性能的，对应@Retention（RetentionPolicy.RUNTIME）。

编译期(Compile time)注解，以及处理编译期注解的手段 APT 和 Javapoet，对应@Retention(RetentionPolicy.CLASS)。 其中 apt+javaPoet 目前也是应用比较广泛，在一些大的开源库，如 EventBus3.0+,页面路由 ARout、Dagger、Retrofit 等均有使用的身影，注解不仅仅是通过反射一种方式来使用，也可以使用 APT 在编译期处理

43、bitmap recycler 相关

在 Android 中，Bitmap 的存储分为两部分，一部分是 Bitmap 的数据，一部分是 Bitmap 的引用。 在 Android2.3 时代，Bitmap 的引用是放在堆中的，而 Bitmap 的数据部分是放在栈中的，需要用户调用 recycle 方法手动进行内存回收，而在 Android2.3 之后，整个 Bitmap，包括数据和引用，都放在了堆中，这样，整个 Bitmap 的回收就全部交给 GC 了，这个 recycle 方法就再也不需要使用了。

bitmap recycler 引发的问题：当图像的旋转角度小余两个像素点之间的夹角时，图像即使旋转也无法显示，因此，系统完全可以认为图像没有发生变化。这时系统就直接引用同一个对象来进行操作，避免内存浪费。

44、强引用置为 null，会不会被回收？

不会立即释放对象占用的内存。 如果对象的引用被置为 null，只是断开了当前线程栈帧中对该对象的引用关系，而 垃圾收集器是运行在后台的线程，只有当用户线程运行到安全点(safe point)或者安全区域才会扫描对象引用关系，扫描到对象没有被引用则会标记对象，这时候仍然不会立即释放该对象内存，因为有些对象是可恢复的（在 finalize 方法中恢复引用 ）。只有确定了对象无法恢复引用的时候才会清除对象内存。

45、Bundle 传递数据为什么需要序列化？

序列化，表示将一个对象转换成可存储或可传输的状态。序列化的原因基本三种情况：

1.永久性保存对象，保存对象的字节序列到本地文件中；

2.对象在网络中传递；

3.对象在 IPC 间传递。

46、广播传输的数据是否有限制，是多少，为什么要限制？

Intent 在传递数据时是有大小限制的，大约限制在 1MB 之内，你用 Intent 传递数据，实际上走的是跨进程通信（IPC），跨进程通信需要把数据从内核 copy 到进程中，每一个进程有一个接收内核数据的缓冲区，默认是 1M；如果一次传递的数据超过限制，就会出现异常。

不同厂商表现不一样有可能是厂商修改了此限制的大小，也可能同样的对象在不同的机器上大小不一样。

传递大数据，不应该用 Intent；考虑使用 ContentProvider 或者直接匿名共享内存。简单情况下可以考虑分段传输。

47、是否了解硬件加速？

硬件加速就是运用 GPU 优秀的运算能力来加快渲染的速度，而通常的基于软件的绘制渲染模式是完全利用 CPU 来完成渲染。

1.硬件加速是从 API 11 引入，API 14 之后才默认开启。对于标准的绘制操作和控件都是支持的，但是对于自定义 View 的时候或者一些特殊的绘制函数就需要考虑是否需要关闭硬件加速。

2.我们面对不支持硬件加速的情况，就需要限制硬件加速，这个兼容性的问题是因为硬件加速是把 View 的绘制函数转化为使用 OpenGL 的函数来进完成实际的绘制的，那么必然会存在 OpenGL 中不支持原始回执函数的情况，对于这些绘制函数，就会失效。

3.硬件加速的消耗问题，因为是使用 OpenGL，需要把系统中 OpenGL 加载到内存中，OpenGL API 调用就会占用 8MB，而实际上会占用更多内存，并且使用了硬件必然增加耗电量了。

4.硬件加速的优势还有 display list 的设计，使用这个我们不需要每次重绘都执行大量的代码，基于软件的绘制模式会重绘脏区域内的所有控件，而 display 只会更新列表，然后绘制列表内的控件。

1. CPU 更擅长复杂逻辑控制，而 GPU 得益于大量 ALU 和并行结构设计，更擅长数学运算。

48、ContentProvider 的权限管理(读写分离，权限控制-精确到表级，URL 控制)。

对于 ContentProvider 暴露出来的数据，应该是存储在自己应用内存中的数据，对于一些存储在外部存储器上的数据，并不能限制访问权限，使用 ContentProvider 就没有意义了。对于 ContentProvider 而言，有很多权限控制，可以在 AndroidManifest.xml 文件中对节点的属性进行配置，一般使用如下一些属性设置：

- android:grantUriPermssions:临时许可标志。
- android:permission:Provider 读写权限。
- android:readPermission:Provider 的读权限。
- android:writePermission:Provider 的写权限。
- android:enabled:标记允许系统启动 Provider。
- android:exported:标记允许其他应用程序使用这个 Provider。
- android:multiProcess:标记允许系统启动 Provider 相同的进程中调用客户端。

49、Fragment 状态保存

Fragment 状态保存入口:

1、Activity 的状态保存, 在 Activity 的 onSaveInstanceState()里, 调用了 FragmentManger 的 saveAllState()方法, 其中会对 mActive 中各个 Fragment 的实例状态和 View 状态分别进行保存.

2、FragmentManager 还提供了 public 方法: saveFragmentInstanceState(), 可以对单个 Fragment 进行状态保存, 这是提供给我们用的。

3、FragmentManager 的 moveToState()方法中, 当状态回退到 ACTIVITY_CREATED, 会调用 saveFragmentViewState()方法, 保存 View 的状态.

50、直接在 Activity 中创建一个 thread 跟在 service 中创建一个 thread 之间的区别?

在 Activity 中被创建：该 Thread 的就是为这个 Activity 服务的，完成这个特定的 Activity 交代的任务，主动通知该 Activity 一些消息和事件，Activity 销毁后，该 Thread 也没有存活的意义了。

在 Service 中被创建：这是保证最长生命周期的 Thread 的唯一方式，只要整个 Service 不退出，Thread 就可以一直在后台执行，一般在 Service 的 onCreate() 中创建，在 onDestroy()中销毁。所以，在 Service 中创建的 Thread，适合长期执行一些独立于 APP 的后台任务，比较常见的就是：在 Service 中保持与服务器端的长连接。

51、如何计算一个 Bitmap 占用内存的大小，怎么保证加载 Bitmap 不产生内存溢出？

```
Bitamp 占用内存大小 = 宽度像素 x （inTargetDensity / inDensity） x 高度像素 x （inTargetDensity / inDensity）x 一个像素所占的内存
```

注：这里 inDensity 表示目标图片的 dpi（放在哪个资源文件夹下），inTargetDensity 表示目标屏幕的 dpi，所以你可以发现 inDensity 和 inTargetDensity 会对 Bitmap 的宽高进行拉伸，进而改变 Bitmap 占用内存的大小。

在 Bitmap 里有两个获取内存占用大小的方法。

getByteCount()：API12 加入，代表存储 Bitmap 的像素需要的最少内存。

getAllocationByteCount()：API19 加入，代表在内存中为 Bitmap 分配的内存大小，代替了 getByteCount() 方法。 在不复用 Bitmap 时，getByteCount() 和 getAllocationByteCount 返回的结果是一样的。在通过复用 Bitmap 来解码图片时，那么 getByteCount() 表示新解码图片占用内存的大 小，

getAllocationByteCount() 表示被复用 Bitmap 真实占用的内存大小（即 mBuffer 的长度）。

为了保证在加载 Bitmap 的时候不产生内存溢出，可以使用 BitmapFactory 进行图片压缩，主要有以下几个参数：

BitmapFactory.Options.inPreferredConfig：将 ARGB_8888 改为 RGB_565，改变编码方式，节约内存。 BitmapFactory.Options.inSampleSize：缩放比例，可以参考 Luban 那个库，根据图片宽高计算出合适的缩放比例。

BitmapFactory.Options.inPurgeable：让系统可以内存不足时回收内存。

52、对于应用更新这块是如何做的？（灰度，强制更新，分区域更新）

1、通过接口获取线上版本号，versionCode 2、比较线上的 versionCode 和本地的 versionCode，弹出更新窗口 3、下载 APK 文件（文件下载） 4、安装 APK

灰度： (1)找单一渠道投放特别版本。 (2)做升级平台的改造，允许针对部分用户推送升级通知甚至版本强制升级。 (3)开放单独的下载入口。 (4)是两个版本的代码都打到 app 包里，然后在 app 端植入测试框架，用来控制显示哪个版本。测试框架负责与服务器端 api 通信，由服务器端控制 app 上 A/B 版本的分布，可以实现指定的一组用户看到 A 版本，其它用户看到 B 版本。服务端会有相应的报表来显示 A/B 版本的数量和效果对比。最后可以由服务端的后台来控制，全部用户在线切换到 A 或者 B 版本~

无论哪种方法都需要做好版本管理工作，分配特别的版本号以示区别。 当然，既然是做灰度，数据监控（常规数据、新特性数据、主要业务数据）还是要做到位，该打的数据桩要打。 还有，灰度版最好有收回的能力，一般就是强制升级下一个正式版。

强制更新:一般的处理就是进入应用就弹窗通知用户有版本更新，弹窗可以没有取消按钮并不能取消。这样用户就只能选择更新或者关闭应用了，当然也可以添加取消按钮，但是如果用户选择取消则直接退出应用。

增量更新：bsdiff：二进制差分工具 bsdiff 是相应的补丁合成工具,根据两个不同版本的二进制文件，生成补丁文件.patch 文件。通过 bspatch 使旧的 apk 文件与不定文件合成新的 apk。 注意通过 apk 文件的 md5 值进行区分版本。

53、请解释安卓为啥要加签名机制。

1、发送者的身份认证 由于开发商可能通过使用相同的 Package Name 来混淆替换已经安装的程序，以此保证签名不同的包不被替换。

2、保证信息传输的完整性 签名对于包中的每个文件进行处理，以此确保包中内容不被替换。

3、防止交易中的抵赖发生， Market 对软件的要求。

54、为什么 bindService 可以跟 Activity 生命周期联动？

1、bindService 方法执行时，LoadedApk 会记录 ServiceConnection 信息。

2、Activity 执行 finish 方法时，会通过 LoadedApk 检查 Activity 是否存在未注销/解绑的 BroadcastReceiver 和 ServiceConnection，如果有，那么会通知 AMS 注销/解绑对应的 BroadcastReceiver 和 Service，并打印异常信息，告诉用户应该主动执行注销/解绑的操作。

55、如何通过 Gradle 配置多渠道包？

用于生成不同渠道的包

```
android {

    productFlavors {
```

```
        xiaomi {}
        baidu {}
        wandoujia {}
        _360 {}        // 或“"360"{}”，数字需下划线开头或加上双引号
    }
}
```

执行./gradlew assembleRelease ，将会打出所有渠道的 release 包；

执行./gradlew assembleWandoujia，将会打出豌豆荚渠道的 release 和 debug 版的包；

执行./gradlew assembleWandoujiaRelease 将生成豌豆荚的 release 包。

因此，可以结合 buildType 和 productFlavor 生成不同的 Build Variants，即类型与渠道不同的组合。

56、activty 和 Fragmengt 之间怎么通信，Fragmengt 和 Fragmengt 怎么通信？

（一）Handler

（二）广播

（三）事件总线：EventBus、RxBus、Otto

（四）接口回调

（五）Bundle 和 setArguments(bundle)

57、自定义 view 效率高于 xml 定义吗？说明理由。

自定义 view 效率高于 xml 定义：

1、少了解析 xml。

2.、自定义 View 减少了 ViewGroup 与 View 之间的测量,包括父量子,子量自身,

子在父中位置摆放,当子 view 变化时,父的某些属性都会跟着变化。

58、广播注册一般有几种，各有什么优缺点？

第一种是常驻型(静态注册)：当应用程序关闭后如果有信息广播来，程序也会被系统调用，自己运行。

第二种不常驻(动态注册)：广播会跟随程序的生命周期。

动态注册

优点： 在 android 的广播机制中，动态注册优先级高于静态注册优先级，因此在必要情况下，是需要动态注册广播接收者的。

缺点： 当用来注册的 Activity 关掉后，广播也就失效了。

静态注册

优点： 无需担忧广播接收器是否被关闭，只要设备是开启状态，广播接收器就是打开着的。

59、服务启动一般有几种，服务和 activty 之间怎么通信，服务和服务之间怎么通信

方式：

1、startService：

onCreate()--->onStartCommand() ---> onDestory()

如果服务已经开启，不会重复的执行 onCreate()， 而是会调用

onStartCommand()。一旦服务开启跟调用者(开启者)就没有任何关系了。 开启者退出了，开启者挂了，服务还在后台长期的运行。 开启者不能调用服务里面的方法。

2、bindService：

onCreate() --->onBind()--->onunbind()--->onDestory()

bind 的方式开启服务，绑定服务，调用者挂了，服务也会跟着挂掉。 绑定者可以调用服务里面的方法。

通信：

1、通过 Binder 对象。

2、通过 broadcast(广播)。

60、ddms 和 traceView 的区别？

ddms 原意是：davik debug monitor service。简单的说 ddms 是一个程序执行查看器，在里面可以看见线程和堆栈等信息，traceView 是程序性能分析器。traceview 是 ddms 中的一部分内容。

Traceview 是 Android 平台特有的数据采集和分析工具，它主要用于分析 Android 中应用程序的 hotspot（瓶颈）。Traceview 本身只是一个数据分析工具，而数据的采集则需要使用 Android SDK 中的 Debug 类或者利用 DDMS 工具。二者的用法如下：开发者在一些关键代码段开始前调用 Android SDK 中 Debug 类的 startMethodTracing 函数，并在关键代码段结束前调用 stopMethodTracing 函数。这两个函数运行过程中将采集运行时间内该应用所有线程（注意，只能是 Java 线程） 的函数执行情况， 并将采集数据保存到 /mnt/sdcard/下的一个文件中。 开发者然后需要利用 SDK 中的 Traceview 工具来分析这些数据。

61、ListView 卡顿原因

Adapter 的 getView 方法里面 convertView 没有使用 setTag 和 getTag 方式；

在 getView 方法里面 ViewHolder 初始化后的赋值或者是多个控件的显示状态和背景的显示没有优化好，抑或是里面含有复杂的计算和耗时操作；

在 getView 方法里面 inflate 的 row 嵌套太深（布局过于复杂）或者是布局里面有大图片或者背景所致；

Adapter 多余或者不合理的 notifySetDataChanged；

listview 被多层嵌套，多次的 onMessure 导致卡顿，如果多层嵌套无法避免，建议把 listview 的高和宽设置为 match_parent. 如果是代码继承的 listview，那么也请你别忘记为你的继承类添加上 LayoutPrams，注意高和宽都 mactch_parent 的；

62、AndroidManifest 的作用与理解

AndroidManifest.xml 文件，也叫清单文件，来获知应用中是否包含该组件，如果有会直接启动该组件。可以理解是一个应用的配置文件。

作用：

- 为应用的 Java 软件包命名。软件包名称充当应用的唯一标识符。
- 描述应用的各个组件，包括构成应用的 Activity、服务、广播接收器和内容提供程序。它还为实现每个组件的类命名并发布其功能，例如它们可以处理的 Intent - 消息。这些声明向 Android 系统告知有关组件以及可以启动这些组件的条件的信息。
- 确定托管应用组件的进程。
- 声明应用必须具备哪些权限才能访问 API 中受保护的部分并与其他应用交互。还声明其他应用与该应用组件交互所需具备的权限
- 列出 Instrumentation 类，这些类可在应用运行时提供分析和其他信息。这些声明只会在应用处于开发阶段时出现在清单中，在应用发布之前将移除。
- 声明应用所需的最低 Android API 级别

- 列出应用必须链接到的库

63、LaunchMode 应用场景

standard，创建一个新的 Activity。

singleTop，栈顶不是该类型的 Activity，创建一个新的 Activity。否则，onNewIntent。

singleTask，回退栈中没有该类型的 Activity，创建 Activity，否则，onNewIntent+ClearTop。

注意:

设置了"singleTask"启动模式的 Activity，它在启动的时候，会先在系统中查找属性值 affinity 等于它的属性值 taskAffinity 的 Task 存在；如果存在这样的 Task，它就会在这个 Task 中启动，否则就会在新的任务栈中启动。因此， 如果我们想要设置了"singleTask"启动模式的 Activity 在新的任务中启动，就要为它设置一个独立的 taskAffinity 属性值。

如果设置了"singleTask"启动模式的 Activity 不是在新的任务中启动时，它会在已有的任务中查看是否已经存在相应的 Activity 实例， 如果存在，就会把位于这个 Activity 实例上面的 Activity 全部结束掉，即最终这个 Activity 实例会位于任务的 Stack 顶端中。

在一个任务栈中只有一个”singleTask”启动模式的 Activity 存在。他的上面可以有其他的 Activity。这点与 singleInstance 是有区别的。

singleInstance，回退栈中，只有这一个 Activity，没有其他 Activity。

singleTop 适合接收通知启动的内容显示页面。

例如，某个新闻客户端的新闻内容页面，如果收到 10 个新闻推送，每次都打开一个新闻内容页面是很烦人的。

singleTask 适合作为程序入口点。

例如浏览器的主界面。不管从多少个应用启动浏览器，只会启动主界面一次，其余情况都会走 onNewIntent，并且会清空主界面上面的其他页面。

singleInstance 应用场景：

闹铃的响铃界面。 你以前设置了一个闹铃：上午 6 点。在上午 5 点 58 分，你启动了闹铃设置界面，并按 Home 键回桌面；在上午 5 点 59 分时，你在微信和朋友聊天；在 6 点时，闹铃响了，并且弹出了一个对话框形式的 Activity(名为 AlarmAlertActivity) 提示你到 6 点了(这个 Activity 就是以 SingleInstance 加载模式打开的)，你按返回键，回到的是微信的聊天界面，这是因为 AlarmAlertActivity 所在的 Task 的栈只有他一个元素， 因此退出之后这个 Task 的栈空了。如果是以 SingleTask 打开 AlarmAlertActivity，那么当闹铃响了的时候，按返回键应该进入闹铃设置界面。

64、说说 Activity、Intent、Service 是什么关系

他们都是 Android 开发中使用频率最高的类。其中 Activity 和 Service 都是 Android 四大组件之一。他俩都是 Context 类的子类 ContextWrapper 的子类，因此他俩可以算是兄弟关系吧。不过兄弟俩各有各自的本领，Activity 负责用户界面的显示和交互，Service 负责后台任务的处理。Activity 和 Service 之间可以通过 Intent 传递数据，因此 可以把 Intent 看作是通信使者。

65、ApplicationContext 和 ActivityContext 的区别

这是两种不同的 context，也是最常见的两种.第一种中 context 的生命周期与 Application 的生命周期相关的，context 随着 Application 的销毁而销毁，伴随 application 的一生，与 activity 的生命周期无关.第二种中的 context 跟 Activity 的生命周期是相关的，但是对一个 Application 来说，Activity 可以销毁几次，那么属于 Activity 的 context 就会销毁多次.至于用哪种 context，得看应用场景。

还有就是，在使用 context 的时候，小心内存泄露，防止内存泄露，注意一下几个方面：

- 不要让生命周期长的对象引用 activity context，即保证引用 activity 的对象要与 activity 本身生命周期是一样的。
- 对于生命周期长的对象，可以使用 application context。
- 避免非静态的内部类，尽量使用静态类，避免生命周期问题，注意内部类对外部对象引用导致的生命周期变化。

66、Handler、Thread 和 HandlerThread 的差别

1、Handler：在 android 中负责发送和处理消息，通过它可以实现其他支线线程与主线程之间的消息通讯。

2、Thread：Java 进程中执行运算的最小单位，亦即执行处理机调度的基本单位。某一进程中一路单独运行的程序。

3、HandlerThread：一个继承自 Thread 的类 HandlerThread，Android 中没有对 Java 中的 Thread 进行任何封装，而是提供了一个继承自 Thread 的类 HandlerThread 类，这个类对 Java 的 Thread 做了很多便利的封装。

HandlerThread 继承于 Thread，所以它本质就是个 Thread。与普通 Thread 的差别就在于，它在内部直接实现了 Looper 的实现，这是 Handler 消息机制必不可

少的。有了自己的 looper，可以让我们在自己的线程中分发和处理消息。如果不用 HandlerThread 的话，需要手动去调用 Looper.prepare()和 Looper.loop()这些方法。

67、ThreadLocal 的原理

ThreadLocal 是一个关于创建线程局部变量的类。使用场景如下所示：

> 实现单个线程单例以及单个线程上下文信息存储，比如交易 id 等。
>
> 实现线程安全，非线程安全的对象使用 ThreadLocal 之后就会变得线程安全，因为每个线程都会有一个对应的实例。 承载一些线程相关的数据，避免在方法中来回传递参数。

当需要使用多线程时，有个变量恰巧不需要共享，此时就不必使用 synchronized 这么麻烦的关键字来锁住，每个线程都相当于在堆内存中开辟一个空间，线程中带有对共享变量的缓冲区，通过缓冲区将堆内存中的共享变量进行读取和操作，ThreadLocal 相当于线程内的内存，一个局部变量。每次可以对线程自身的数据读取和操作，并不需要通过缓冲区与 主内存中的变量进行交互。并不会像 synchronized 那样修改主内存的数据，再将主内存的数据复制到线程内的工作内存。ThreadLocal 可以让线程独占资源，存储于线程内部，避免线程堵塞造成 CPU 吞吐下降。

在每个 Thread 中包含一个 ThreadLocalMap，ThreadLocalMap 的 key 是 ThreadLocal 的对象，value 是独享数据。

68、计算一个 view 的嵌套层级

```
private int getParents(ViewParents view){
    if(view.getParents() == null)
        return 0;
    } else {
```

```
    return (1 + getParents(view.getParents));
   }
}
```

69、MVP，MVVM，MVC 解释和实践

MVC:

- 视图层(View) 对应于 xml 布局文件和 java 代码动态 view 部分
- 控制层(Controller) MVC 中 Android 的控制层是由 Activity 来承担的，Activity 本来主要是作为初始化页面，展示数据的操作，但是因为 XML 视图功能太弱，所以 Activity 既要负责视图的显示又要加入控制逻辑，承担的功能过多。
- 模型层(Model) 针对业务模型，建立数据结构和相关的类，它主要负责网络请求，数据库处理，I/O 的操作。

总结

具有一定的分层，model 彻底解耦，controller 和 view 并没有解耦 层与层之间的交互尽量使用回调或者去使用消息机制去完成，尽量避免直接持有 controller 和 view 在 android 中无法做到彻底分离，但在代码逻辑层面一定要分清 业务逻辑被放置在 model 层，能够更好的复用和修改增加业务。

MVP

通过引入接口 BaseView，让相应的视图组件如 Activity，Fragment 去实现 BaseView，实现了视图层的独立，通过中间层 Preseter 实现了 Model 和 View 的完全解耦。MVP 彻底解决了 MVC 中 View 和 Controller 傻傻分不清楚的问题，

但是随着业务逻辑的增加，一个页面可能会非常复杂，UI 的改变是非常多，会有非常多的 case，这样就会造成 View 的接口会很庞大。

MVVM

MVP 中我们说过随着业务逻辑的增加，UI 的改变多的情况下，会有非常多的跟 UI 相关的 case，这样就会造成 View 的接口会很庞大。而 MVVM 就解决了这个问题，通过双向绑定的机制，实现数据和 UI 内容，只要想改其中一方，另一方都能够及时更新的一种设计理念，这样就省去了很多在 View 层中写很多 case 的情况，只需要改变数据就行。

MVVM 与 DataBinding 的关系？

MVVM 是一种思想，DataBinding 是谷歌推出的方便实现 MVVM 的工具。

看起来 MVVM 很好的解决了 MVC 和 MVP 的不足，但是由于数据和视图的双向绑定，导致出现问题时不太好定位来源，有可能数据问题导致，也有可能业务逻辑中对视图属性的修改导致。如果项目中打算用 MVVM 的话可以考虑使用官方的架构组件 ViewModel、LiveData、DataBinding 去实现 MVVM。

三者如何选择？

- 如果项目简单，没什么复杂性，未来改动也不大的话，那就不要用设计模式或者架构方法，只需要将每个模块封装好，方便调用即可，不要为了使用设计模式或架构方法而使用。
- 对于偏向展示型的 app，绝大多数业务逻辑都在后端，app 主要功能就是展示数据，交互等，建议使用 mvvm。
- 对于工具类或者需要写很多业务逻辑 app，使用 mvp 或者 mvvm 都可。

70、SharedPrefrences 的 apply 和 commit 有什么区别？

这两个方法的区别在于：

apply 没有返回值而 commit 返回 boolean 表明修改是否提交成功。

apply 是将修改数据原子提交到内存, 而后异步真正提交到硬件磁盘, 而 commit 是同步的提交到硬件磁盘，因此，在多个并发的提交 commit 的时候，他们会等待正在处理的 commit 保存到磁盘后在操作，从而降低了效率。而 apply 只是原子的提交到内容，后面有调用 apply 的函数的将会直接覆盖前面的内存数据，这样从一定程度上提高了很多效率。

apply 方法不会提示任何失败的提示。 由于在一个进程中，sharedPreference 是单实例，一般不会出现并发冲突，如果对提交的结果不关心的话，建议使用 apply，当然需要确保提交成功且有后续操作的话，还是需要用 commit 的。

71、Base64、MD5 是加密方法么？

Base64 是什么？

Base64 是用文本表示二进制的编码方式，它使用 4 个字节的文本来表示 3 个字节的原始二进制数据。它将二进制数据转换成一个由 64 个可打印的字符组成的序列：A-Za-z0-9+/

MD5 是什么？

MD5 是哈希算法的一种，可以将任意数据产生出一个 128 位（16 字节）的散列值，用于确保信息传输完整一致。我们常在注册登录模块使用 MD5，用户密码可以使用 MD5 加密的方式进行存储。如：md5(hello world,32) = 5eb63bbbe01eeed093cb22bb8f5acdc3

加密，指的是对数据进行转换以后，数据变成了另一种格式，并且除了拿到解密方法的人，没人能把数据转换回来。 MD5 是一种信息摘要算法，它是不可逆的，不可以解密。所以它只能算的上是一种单向加密算法。 Base64 也不是加密算法，它是一种数据编码方式，虽然是可逆的，但是它的编码方式是公开的，无所谓加密。

72、HttpClient 和 HttpConnection 的区别？

Http Client 适用于 web 浏览器，拥有大量灵活的 API，实现起来比较稳定，且其功能比较丰富，提供了很多工具，封装了 http 的请求头，参数，内容体，响应，还有一些高级功能，代理、COOKIE、鉴权、压缩、连接池的处理。 但是，正因此，在不破坏兼容性的前提下，其庞大的 API 也使人难以改进，因此 Android 团队对于修改优化 Apache Http Client 并不积极。(并在 Android 6.0 中抛弃了 Http Client，替换成 OkHttp)

HttpURLConnection 对于大部分功能都进行了包装，Http Client 的高级功能代码会较复杂，另外，HttpURLConnection 在 Android 2.3 中增加了一些 Https 方面的改进(包括 Http Client，两者都支持 https)。且在 Android 4.0 中增加了 response cache。当缓存被安装后(调用 HttpResponseCache 的 install()方法)，所有的 HTTP 请求都会满足以下三种情况：

- 所有的缓存响应都由本地存储来提供。因为没有必要去发起任务的网络连接请求，所有的响应都可以立刻获取到。
- 视情况而定的缓存响应必须要有服务器来进行更新检查。比如说客户端发起了一条类似于“如果/foo.png 这张图片发生了改变，就将它发送给我”这样的请求，服务器需要将更新后的数据进行返回，或者返回一个 304 Not Modified 状态。如果请求的内容没有发生，客户端就不会下载任何数据。

- 没有缓存的响应都是由服务器直接提供的。这部分响应会在稍后存储到响应缓存中。

在 Android 2.2 版本之前，HttpClient 拥有较少的 bug，因此使用它是最好的选择。 而在 Android 2.3 版本及以后，HttpURLConnection 则是最佳的选择。它的 API 简单，体积较小，因而非常适用于 Android 项目。压缩和缓存机制可以有效地减少网络访问的流量，在提升速度和省电方面也起到了较大的作用。对于新的应用程序应该更加偏向于使用 HttpURLConnection，因为在以后的工作当中 Android 官方也会将更多的时间放在优化 HttpURLConnection 上面。

73、ActivityA 跳转 ActivityB 然后 B 按 back 返回 A，各自的生命周期顺序，A 与 B 均不透明。

ActivityA 跳转到 ActivityB：

```
Activity A: onPause

Activity B: onCreate

Activity B: onStart

Activity B: onResume

Activity A: onStop
```

ActivityB 返回 ActivityA：

```
Activity B: onPause

Activity A: onRestart

Activity A: onStart

Activity A: onResume

Activity B: onStop

Activity B: onDestroy
```

74、如何通过广播拦截和 abort 一条短信？

可以监听这条信号，在传递给真正的接收程序时，我们将自定义的广播接收程序的优先级大于它，并且取消广播的传播，这样就可以实现拦截短信的功能了。

75、BroadcastReceiver，LocalBroadcastReceiver 区别？

1、应用场景

1、BroadcastReceiver 用于应用之间的传递消息；

2、而 LocalBroadcastManager 用于应用内部传递消息，比 broadcastReceiver 更加高效。

2、安全

1、BroadcastReceiver 使用的 Content API，所以本质上它是跨应用的，所以在使用它时必须要考虑到不要被别的应用滥用；

2、LocalBroadcastManager 不需要考虑安全问题，因为它只在应用内部有效。

3、原理方面

(1) 与 BroadcastReceiver 是以 Binder 通讯方式为底层实现的机制不同，

LocalBroadcastManager 的核心实现实际还是 Handler，只是利用到了

IntentFilter 的 match 功能，至于 BroadcastReceiver 换成其他接口也无所谓，顺便利用了现成的类和概念而已。

(2) LocalBroadcastManager 因为是 Handler 实现的应用内的通信，自然安全性更好，效率更高。

76、如何选择第三方，从那些方面考虑？

大方向：从软件环境做判断

性能是开源软件第一解决的问题。

一个好的生态，是一个优秀的开源库必备的，取决标准就是观察它是否一直在持续更新迭代，是否能及时处理 github 上用户提出来的问题。大家在社区针对这个开源库是否有比较活跃的探讨。

背景，该开源库由谁推出，由哪个公司推出来的。

用户数和有哪些知名的企业落地使用

小方向：从软件开发者的角度做判断

是否解决了我们现有问题或长期来看带来的维护成本。

公司有多少人会。

学习成本。

77、简单说下接入支付的流程，是否自己接入过支付功能？

Alipay 支付功能：

1.首先登录支付宝开放平台创建应用，并给应用添加 App 支付功能， 由于 App 支付功能需要签约，因此需要上传公司信息和证件等资料进行签约。

2.签约成功后，需要配置秘钥。使用支付宝提供的工具生成 RSA 公钥和私钥，公钥需要设置到管理后台。

3.android studio 集成

```
（1）copy jar 包；

（2）发起支付请求，处理支付请求。
```

78、单例实现线程的同步的要求：

1.单例类确保自己只有一个实例(构造函数私有:不被外部实例化,也不被继承)。

2.单例类必须自己创建自己的实例。

3.单例类必须为其他对象提供唯一的实例。

79、如何保证 Service 不被杀死？

Android 进程不死从 3 个层面入手：

A.提供进程优先级，降低进程被杀死的概率

方法一：监控手机锁屏解锁事件，在屏幕锁屏时启动 1 个像素的 Activity，在用户解锁时将 Activity 销毁掉。

方法二：启动前台 service。

方法三：提升 service 优先级：

在 AndroidManifest.xml 文件中对于 intent-filter 可以通过 android:priority = "1000"这个属性设置最高优先级，1000 是最高值，如果数字越小则优先级越低，同时适用于广播。

B. 在进程被杀死后，进行拉活

方法一：注册高频率广播接收器，唤起进程。如网络变化，解锁屏幕，开机等

方法二：双进程相互唤起。

方法三：依靠系统唤起。

方法四：onDestroy 方法里重启 service：service + broadcast 方式，就是当 service 走 ondestory 的时候，发送一个自定义的广播，当收到广播的时候，重新启动 service；

C. 依靠第三方

根据终端不同，在小米手机（包括 MIUI）接入小米推送、华为手机接入华为推送；其他手机可以考虑接入腾讯信鸽或极光推送与小米推送做 A/B Test。

80、说说 ContentProvider、ContentResolver、ContentObserver 之间的关系？

ContentProvider：管理数据，提供数据的增删改查操作，数据源可以是数据库、文件、XML、网络等，ContentProvider 为这些数据的访问提供了统一的接口，可以用来做进程间数据共享。

ContentResolver：ContentResolver 可以为不同 URI 操作不同的 ContentProvider 中的数据，外部进程可以通过 ContentResolver 与 ContentProvider 进行交互。

ContentObserver：观察 ContentProvider 中的数据变化，并将变化通知给外界。

81、如何导入外部数据库?

把原数据库包括在项目源码的 res/raw。

android 系统下数据库应该存放在 /data/data/com.（package name）/ 目录下，所以我们需要做的是把已有的数据库传入那个目录下。操作方法是用 FileInputStream 读取原数据库，再用 FileOutputStream 把读取到的东西写入到那个目录。

82、LinearLayout、FrameLayout、RelativeLayout 性能对比，为什么？

RelativeLayout 会让子 View 调用 2 次 onMeasure，LinearLayout 在有 weight 时，也会调用子 View 2 次 onMeasure

RelativeLayout 的子 View 如果高度和 RelativeLayout 不同，则会引发效率问题，当子 View 很复杂时，这个问题会更加严重。如果可以，尽量使用 padding 代替 margin。

在不影响层级深度的情况下,使用 LinearLayout 和 FrameLayout 而不是 RelativeLayout。

为什么 Google 给开发者默认新建了个 RelativeLayout，而自己却在 DecorView 中用了个

LinearLayout？

因为 DecorView 的层级深度是已知而且固定的，上面一个标题栏，下面一个内容栏。采用 RelativeLayout 并不会降低层级深度，所以此时在根节点上用 LinearLayout 是效率最高的。而之所以给开发者默认新建了个 RelativeLayout 是希望开发者能采用尽量少的 View 层级来表达布局以实现性能最优，因为复杂的 View 嵌套对性能的影响会更大一些。

83、scheme 跳转协议

Android 中的 scheme 是一种页面内跳转协议，通过定义自己的 scheme 协议，可以跳转到 app 中的各个页面

服务器可以定制化告诉 app 跳转哪个页面

App 可以通过跳转到另一个 App 页面

可以通过 H5 页面跳转页面

84、HandlerThread

1、HandlerThread 原理

当系统有多个耗时任务需要执行时，每个任务都会开启个新线程去执行耗时任务，这样会导致系统多次创建和销毁线程，从而影响性能。为了解决这一问题，Google 提出了 HandlerThread，HandlerThread 本质上是一个线程类，它继承了 Thread。HandlerThread 有自己的内部 Looper 对象，可以进行 loopr 循环。通过获取 HandlerThread 的 looper 对象传递给 Handler 对象，可以在 handleMessage()方法中执行异步任务。创建 HandlerThread 后必须先调用 HandlerThread.start()方法，Thread 会先调用 run 方法，创建 Looper 对象。当

有耗时任务进入队列时，则不需要开启新线程，在原有的线程中执行耗时任务即可，否则线程阻塞。它在 Android 中的一个具体的使用场景是 IntentService。由于 HanlderThread 的 run()方法是一个无限循环，因此当明确不需要再使用 HandlerThread 时，可以通过它的 quit 或者 quitSafely 方法来终止线程的执行。

2、HanlderThread 的优缺点

HandlerThread 优点是异步不会堵塞，减少对性能的消耗。

HandlerThread 缺点是不能同时继续进行多任务处理，要等待进行处理，处理效率较低。

HandlerThread 与线程池不同，HandlerThread 是一个串队列，背后只有一个线程。

85、IntentService

IntentService 是一种特殊的 Service，它继承了 Service 并且它是一个抽象类，因此必须创建它的子类才能使用 IntentService。

原理

在实现上，IntentService 封装了 HandlerThread 和 Handler。当 IntentService 被第一次启动时，它的 onCreate()方法会被调用，onCreat()方法会创建一个 HandlerThread，然后使用它的 Looper 来构造一个 Handler 对象 mServiceHandler，这样通过 mServiceHandler 发送的消息最终都会在 HandlerThread 中执行。

生成一个默认的且与主线程互相独立的工作者线程来执行所有传送至 onStartCommand()方法的 Intetnt。

生成一个工作队列来传送 Intent 对象给 onHandleIntent()方法，同一时刻只传送一个 Intent 对象，这样一来，你就不必担心多线程的问题。在所有的请求(Intent)都被执行完以后会自动停止服务，所以，你不需要自己去调用 stopSelf()方法来停止。

该服务提供了一个 onBind()方法的默认实现，它返回 null。

提供了一个 onStartCommand()方法的默认实现，它将 Intent 先传送至工作队列，然后从工作队列中每次取出一个传送至 onHandleIntent()方法，在该方法中对 Intent 做相应的处理。

为什么在 mServiceHandler 的 handleMessage()回调方法中执行完 onHandlerIntent()方法后要使用带参数的 stopSelf()方法？

因为 stopSel()方法会立即停止服务，而 stopSelf（int startId）会等待所有的消息都处理完毕后才终止服务，一般来说，stopSelf(int startId)在尝试停止服务之前会判断最近启动服务的次数是否和 startId 相等，如果相等就立刻停止服务，不相等则不停止服务。

86、如何将一个 Activity 设置成窗口的样式。

中配置：

```
android:theme="@android:style/Theme.Dialog"
```

另外

```
android:theme="@android:style/Theme.Translucnt"
```

是设置透明。

87、Android 中跨进程通讯的几种方式

1：访问其他应用程序的 Activity 如调用系统通话应用

```
Intent callIntent = new Intent(Intent.ACTION_CALL,Uri.parse("tel:12345678");

startActivity(callIntent);
```

2：Content Provider 如访问系统相册

3：广播（Broadcast） 如显示系统时间

4：AIDL 服务

88、显示 Intent 与隐式 Intent 的区别

对明确指出了目标组件名称的 Intent，我们称之为“显式 Intent”。

对于没有明确指出目标组件名称的 Intent，则称之为“隐式 Intent”。

对于隐式意图，在定义 Activity 时，指定一个 intent-filter，当一个隐式意图对象被一个意图过滤器进行匹配时，将有三个方面会被参考到：

动作(Action)

类别(Category ['kætɪg(ə)rɪ])

数据(Data)

```
<activity android:name=".MainActivity"  android:label="@string/app_name">
            <intent-filter>
                <action android:name="com.wpc.test" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="image/gif"/>
            </intent-filter>
</activity>
```

89、Android Holo 主题与 MD 主题的理念，以及你的看法

Holo Theme

Holo Theme 是 Android Design 的最基础的呈现方式。因为是最为基础的 Android Design 呈现形式，每一台 Android 4.X 的手机系统内部都有集成 Holo Theme 需要的控件，即开发者不需要自己设计控件，而是直接从系统里调用相应的控件。在 UI 方面没有任何的亮点，和 Android4.X 的设置/电话的视觉效果极度统一。由此带来的好处显而易见，这个应用作为 Android 应用的辨识度极高，且完全不可能与系统风格产生冲突。

Material Design

Material design 其实是单纯的一种设计语言，它包含了系统界面风格、交互、UI,更加专注拟真,更加大胆丰富的用色,更加丰富的交互形式,更加灵活的布局形式

1.鲜明、形象的界面风格，

2.色彩搭配使得应用看起来非常的大胆、充满色彩感，凸显内容

3.Material design 对于界面的排版非常的重视

4.Material design 的交互设计上采用的是响应式交互，这样的交互设计能把一个应用从简单展现用户所请求的信息，提升至能与用户产生更强烈、更具体化交互的工具。

90、如何让程序自动启动？

定义一个 Braodcastreceiver，action 为 BOOT——COMPLETE，接受到广播后启动程序。

91、Fragment 在 ViewPager 里面的生命周期，滑动 ViewPager 的页面时 Fragment 的生命周期的变化。

92、如何查看模拟器中的 SP 与 SQList 文件。如何可视化查看布局嵌套层数与加载时间。

93、各大平台打包上线的流程与审核时间，常见问题(主流的应用市场说出 3-4 个)

94、屏幕适配的处理技巧都有哪些?

一、为什么要适配

为了保证用户获得一致的用户体验效果,使得某一元素在 Android 不同尺寸、不同分辨率的、不同系统的手机上具备相同的显示效果，能够保持界面上的效果一致,我们需要对各种手机屏幕进行适配!

- Android 系统碎片化：基于 Google 原生系统，小米定制的 MIUI、魅族定制的 flyme、华为定制的 EMUI 等等；
- Android 机型屏幕尺寸碎片化：5 寸、5.5 寸、6 寸等等；
- Android 屏幕分辨率碎片化：320x480、480x800、720x1280、1080x1920 等。

二、基本概念

- 像素（px）：像素就是手机屏幕的最小构成单元，px = 1 像素点 一般情况下 UI 设计师的设计图会以 px 作为统一的计量单位。
- 分辨率：手机在横向、纵向上的像素点数总和 一般描述成 宽*高 ，即横向像素点个数 * 纵向像素点个数（如 1080 x 1920），单位：px。
- 屏幕尺寸：手机对角线的物理尺寸。单位 英寸（inch），一英寸大约 2.54cm 常见的尺寸有 4.7 寸、5 寸、5.5 寸、6 寸。

- 屏幕像素密度（dpi）：每英寸的像素点数，例如每英寸内有 160 个像素点，则其像素密度为 160dpi，单位：dpi（dots per inch）。
- 标准屏幕像素密度（mdpi）：每英寸长度上还有 160 个像素点（160dpi），即称为标准屏幕像素密度（mdpi）。
- 密度无关像素（dp）：与终端上的实际物理像素点无关，可以保证在不同屏幕像素密度的设备上显示相同的效果，是安卓特有的长度单位，dp 与 px 的转换：1dp = （dpi / 160 ） * 1px。
- 独立比例像素（sp）：字体大小专用单位 Android 开发时用此单位设置文字大小，推荐使用 12sp、14sp、18sp、22sp 作为字体大小。

三、适配方案

适配的最多的 3 个分辨率：1280720,19201080,800*480。

解决方案：

对于 Android 的屏幕适配，我认为可以从以下 4 个方面来做：

1、布局组件适配

- 请务必使用密度无关像素 dp 或独立比例像素 sp 单位指定尺寸。
- 使用相对布局或线性布局，不要使用绝对布局
- 使用 wrap_content、match_parent、权重
- 使用 minWidth、minHeight、lines 等属性

dimens 使用：

不同的屏幕尺寸可以定义不同的数值，或者是不同的语言显示我们也可以定义不同的数值，因为翻译后的长度一般都不会跟中文的一致。此外，也可以使用百分比布局或者 AndroidStudio2.2 的新特性约束布局。

2、布局适配

使用限定符（屏幕密度限定符、尺寸限定符、最小宽度限定符、布局别名、屏幕方向限定符)根据屏幕的配置来加载相应的 UI 布局。

3、图片资源适配

使用自动拉伸图.9png 图片格式使图片资源自适应屏幕尺寸。

普通图片和图标：

建议按照官方的密度类型进行切图即可，但一般我们只需 xxhdpi 或 xxxhdpi 的切图即可满足我们的需求；

4、代码适配：

在代码中使用 Google 提供的 API 对设备的屏幕宽度进行测量，然后按照需求进行设置。

5、接口配合：

本地加载图片前判断手机分辨率或像素密度，向服务器请求对应级别图片。

95、断点续传实现？

在本地下载过程中要使用数据库实时存储到底存储到文件的哪个位置了，这样点击开始继续传递时，才能通过 HTTP 的 GET 请求中的 setRequestProperty("Range","bytes=startIndex-endIndex");方法可以告诉服务器，数据从哪里开始，到哪里结束。同时在本地的文件写入时，RandomAccessFile 的 seek()方法也支持在文件中的任意位置进行写入操作。最后通过广播或事件总

线机制将子线程的进度告诉 Activity 的进度条。关于断线续传的 HTTP 状态码是 206，即 HttpStatus.SC_PARTIAL_CONTENT。

96、项目中遇到哪些难题，最终你是如何解决的？

97、Activity 正常和异常情况下的生命周期

98、关于< include >< merge >< stub >三者的使用场景

99、Android 对 HashMap 做了优化后推出的新的容器类是什么？

第二节 Android 高级面试题 （★ ★ ★）

一、性能优化

1、App 稳定性优化

1、你们做了哪些稳定性方面的优化？

随着项目的逐渐成熟，用户基数逐渐增多，DAU 持续升高，我们遇到了很多稳定性方面的问题，对于我们技术同学遇到了很多的挑战，用户经常使用我们的 App 卡顿或者是功能不可用，因此我们就针对稳定性开启了专项的优化，我们主要优化了三项：

- Crash 专项优化（=>2)
- 性能稳定性优化（=>2)
- 业务稳定性优化（=>3)

通过这三方面的优化我们搭建了移动端的高可用平台。同时，也做了很多的措施来让 App 真正地实现了高可用。

2、性能稳定性是怎么做的？

- 全面的性能优化：启动速度、内存优化、绘制优化
- 线下发现问题、优化为主
- 线上监控为主

- Crash 专项优化

我们针对启动速度，内存、布局加载、卡顿、瘦身、流量、电量等多个方面做了多维的优化。

我们的优化主要分为了两个层次，即线上和线下，针对于线下呢，我们侧重于发现问题，直接解决，将问题尽可能在上线之前解决为目的。而真正到了线上呢，我们最主要的目的就是为了监控，对于各个性能纬度的监控呢，可以让我们尽可能早地获取到异常情况的报警。

同时呢，对于线上最严重的性能问题性问题：Crash，我们做了专项的优化，不仅优化了 Crash 的具体指标，而且也尽可能地获取了 Crash 发生时的详细信息，结合后端的聚合、报警等功能，便于我们快速地定位问题。

3、业务稳定性如何保障？

- 数据采集 + 报警
- 需要对项目的主流程与核心路径进行埋点监控，
- 同时还需知道每一步发生了多少异常，这样，我们就知道了所有业务流程的转换率以及相应界面的转换率
- 结合大盘，如果转换率低于某个值，进行报警
- 异常监控 + 单点追查
- 兜底策略

移动端业务高可用它侧重于用户功能完整可用，主要是为了解决一些线上一些异常情况导致用户他虽然没有崩溃，也没有性能问题，但是呢，只是单纯的功能不可用的情况，我们需要对项目的主流程、核心路径进行埋点监控，来计算每一步它真实的转换率是多少，同时呢，还需要知道在每一步到底发生了多少异常。这样我们就知道了所有业务流程的转换率以及相应界面的转换率，有了大盘的数据呢，我们就知道了，如果转换率或者是某些监控的成功率低于某个值，那很有可能就是出现了线上异常，结合了相应的报警功能，我们就不需要等用户来反馈了，这个就是业务稳定性保障的基础。

同时呢，对于一些特殊情况，比如说，开发过程当中或代码中出现了一些 catch 代码块，捕获住了异常，让程序不崩溃，这其实是不合理的，程序虽然没有崩溃，当时程序的功能已经变得不可用，所以呢，这些被 catch 的异常我们也需要上报上来，这样我们才能知道用户到底出现了什么问题而导致的异常。此外，线上还

有一些单点问题，比如说用户点击登录一直进不去，这种就属于单点问题，其实我们是无法找出其和其它问题的共性之处的，所以呢，我们就必须要找到它对应的详细信息。

最后，如果发生了异常情况，我们还采取了一系列措施进行快速止损。（=>4）

4、如果发生了异常情况，怎么快速止损？

- 功能开关
- 统跳中心
- 动态修复：热修复、资源包更新
- 自主修复：安全模式

首先，需要让 App 具备一些高级的能力，我们对于任何要上线的新功能，要加上一个功能的开关，通过配置中心下发的开关呢，来决定是否要显示新功能的入口。如果有异常情况，可以紧急关闭新功能的入口，那就可以让这个 App 处于可控的状态了。

然后，我们需要给 App 设立路由跳转，所有的界面跳转都需要通过路由来分发，

如果我们匹配到需要跳转到有 bug 的这样一个新功能时，那我们就不跳转了，或者是跳转到统一的异常正处理中的界面。如果这两种方式都不可以，那就可以考虑通过热修复的方式来动态修复，目前热修复的方案其实已经比较成熟了，我们完全可以低成本地在我们的项目中添加热修复的能力，当然，如果有些功能是由 RN 或 WeeX 来实现就更好了，那就可以通过更新资源包的方式来实现动态更新。而这些如果都不可以的话呢，那就可以考虑自己去给应用加上一个自主修复的能力，如果 App 启动多次的话，那就可以考虑清空所有的缓存数据，将 App 重置到安装的状态，到了最严重的等级呢，可以阻塞主线程，此时一定要等 App 热修复成功之后才允许用户进入。

需要更全面更深入的理解请查看深入探索 Android 稳定性优化

2、App 启动速度优化

1、启动优化是怎么做的？

- 分析现状、确认问题
- 针对性优化（先概括，引导其深入）
- 长期保持优化效果

在某一个版本之后呢，我们会发现这个启动速度变得特别慢，同时用户给我们的反馈也越来越多，所以，我们开始考虑对应用的启动速度来进行优化。然后，我们就对启动的代码进行了代码层面的梳理，我们发现应用的启动流程已经非常复杂，接着，我们通过一系列的工具来确认是否在主线程中执行了太多的耗时操作。

我们经过了细查代码之后，发现应用主线程中的任务太多，我们就想了一个方案去针对性地解决，也就是进行异步初始化。（引导=>第 2 题） 然后，我们还发现了另外一个问题，也可以进行针对性的优化，就是在我们的初始化代码当中有些的优先级并不是那么高，它可以不放在 Application 的 onCreate 中执行，而完全可以放在之后延迟执行的，因为我们对这些代码进行了延迟初始化，最后，我们还结合了 idealHandler 做了一个更优的延迟初始化的方案，利用它可以在主线程的空闲时间进行初始化，以减少启动耗时导致的卡顿现象。做完这些之后，我们的启动速度就变得很快了。

最后，我简单说下我们是怎么长期来保持启动优化的效果的。首先，我们做了我们的启动器，并且结合了我们的 CI，在线上加上了很多方面的监控。（引导=> 第 4 题）

2、是怎么异步的，异步遇到问题没有？

- 体现演进过程
- 详细介绍启动器

我们最初是采用的普通的一个异步的方案，即 new Thread + 设置线程优先级为后台线程的方式在 Application 的 onCreate 方法中进行异步初始化，后来，我们使用了线程池、IntentService 的方式，但是，在我们应用的演进过程当中，发现代码会变得不够优雅，并且有些场景非常不好处理，比如说多个初始化任务直接的依赖关系，比如说某一个初始化任务需要在某一个特定的生命周期中初始化完成，这些都是使用线程池、IntentService 无法实现的。所以说，我们就开始思考一个新的解决方案，它能够完美地解决我们刚刚所遇到的这些问题。

这个方案就是我们目前所使用的启动器，在启动器的概念中，我们将每一个初始化代码抽象成了一个 Task，然后，对它们进行了一个排序，根据它们之间的依赖关系排了一个有向无环图，接着，使用一个异步队列进行执行，并且这个异步队列它和 CPU 的核心数是强烈相关的，它能够最大程度地保证我们的主线程和别的线程都能够执行我们的任务，也就是大家几乎都可以同时完成。

3、启动优化有哪些容易忽略的注意点？

- cpu time 与 wall time
- 注意延迟初始化的优化
- 介绍下黑科技

首先，在 CPU Profiler 和 Systrace 中有两个很重要的指标，即 cpu time 与 wall time，我们必须清楚 cpu time 与 wall time 之间的区别，wall time 指的是代码执行的时间，而 cpu time 指的是代码消耗 CPU 的时间，锁冲突会造成两者时间差距过大。我们需要以 cpu time 来作为我们优化的一个方向。

其次，我们不仅只追求启动速度上的一个提升，也需要注意延迟初始化的一个优化，对于延迟初始化，通常的做法是在界面显示之后才去进行加载，但是如果此时界面需要进行滑动等与用户交互的一系列操作，就会有很严重的卡顿现象，因此我们使用了 idealHandler 来实现 cpu 空闲时间来执行耗时任务，这极大地提升了用户的体验，避免了因启动耗时任务而导致的页面卡顿现象。

最后，对于启动优化，还有一些黑科技，首先，就是我们采用了类预先加载的方式，我们在 MultiDex.install 方法之后起了一个线程，然后用 Class.forName 的方式来预先触发类的加载，然后当我们这个类真正被使用的时候，就不用再进行类加载的过程了。同时，我们再看 Systrace 图的时候，有一部分手机其实并没有给我们应用去跑满 cpu，比如说它有 8 核，但是却只给了我们 4 核等这些情况，然后，有些应用对此做了一些黑科技，它会将 cpu 的核心数以及 cpu 的频率在启动的时候去进行一个暴力的提升。

4、版本迭代导致的启动变慢有好的解决方式吗？

- 启动器
- 结合 CI
- 监控完善

这种问题其实我们之前也遇到过，这的确非常难以解决。但是，我们后面对此进行了反复的思考与尝试，终于找到了一个比较好的解决方式。

首先，我们使用了启动器去管理每一个初始化任务，并且启动器中每一个任务的执行都是被其自动进行分配的，也就是说这些自动分配的 task 我们会尽量保证它会平均分配在我们每一个线程当中的，这和我们普通的异步是不一样的，它可以很好地缓解我们应用的启动变慢。

其次，我们还结合了 CI，比如说，我们现在限制了一些类，如 Application，如果有人修改了它，我们不会让这部分代码合并到主干分支或者是修改之后会有一些内部的工具如邮件的形式发送到我，然后，我就会和他确认他加的这些代码到底是耗时多少，能否异步初始化，不能异步的话就考虑延迟初始化，如果初始化时间太长，则可以考虑是否能进行懒加载，等用到的时候再去使用等等。

然后，我们会将问题尽可能地暴露在上线之前。同时，我们真正已经到了线上的一个环境下时，我们进行了监控的一个完善，我们不仅是监控了 App 的整个的启动时间，同时呢，我们也将每一个生命周期都进行了一个监控。比如说 Application 的 onCreate 与 onAttachBaseContext 方法的耗时，以及这两个生命周期之间间隔的时间，我们都进行了一个监控，如果说下一次我们发现了这个启动速度变慢了，我们就可以去查找到底是哪一个环节变慢了，我们会和以前的版本进行对比，对比完成之后呢，我们就可以来找这一段新加的代码。

5、开放问题：如果提高启动速度，设计一个延迟加载框架或者 sdk 的方法和注意的问题

需要更全面更深入的理解请查看深入探索 Android 启动速度优化

3、App 内存优化

1、你们内存优化项目的过程是怎么做的？

1、分析现状、确认问题

我们发现我们的 APP 在内存方面可能存在很大的问题，第一方面的原因是我们的线上的 OOM 率比较高。第二点呢，我们经常会看到在我们的 Android Studio 的 Profiler 工具中内存的抖动比较频繁。这是我一个初步的现状，然后在我们知道了这个初步的现状之后，进行了问题的确认，我们经过一系列的调研以及深入研究，我们最终发现我们的项目中存在以下几点大问题，比如说：内存抖动、内存溢出、内存泄漏，还有我们的 Bitmap 使用非常粗犷。

2、针对性优化

比如内存抖动的解决 -> Memory Profiler 工具的使用（呈现了锯齿张图形） -> 分析到具体代码存在的问题（频繁被调用的方法中出现了日志字符串的拼接），也可以说说内存泄漏或内存溢出的解决。

3、效率提升

为了不增加业务同学的工作量，我们使用了一些工具类或 ARTHook 这样的大图检测方案,没有任何的侵入性,同时,我们将这些技术教给了大家,然后让大家一起进行工作效率上的提升。

我们对内存优化工具 Memory Profiler、MAT 的使用比较熟悉，因此针对一系列不同问题的情况，我们写了一系列解决方案的文档，分享给大家。这样，我们整个团队成员的内存优化意识就变强了。

2、你做了内存优化最大的感受是什么？

1、磨刀不误砍柴工

我们一开始并没有直接去分析项目中代码哪些地方存在内存问题，而是先去学习了 Google 官方的一些文档，比如说学习了 Memory Profiler 工具的使用、学习了 MAT 工具的使用，在我们将这些工具学习熟练之后，当在我们的项目中遇到内存问题时，我们就能够很快地进行排查定位问题进行解决。

2、技术优化必须结合业务代码

一开始，我们做了整体 APP 运行阶段的一个内存上报，然后，我们在一些重点的内存消耗模块进行了一些监控，但是后面发现这些监控并没有紧密地结合我们的业务代码，比如说在梳理完项目之后，发现我们项目中存在使用多个图片库的情况，多个图片库的内存缓存肯定是不公用的，所以导致我们整个项目的内存使用量非常高。所以进行技术优化时必须结合我们的业务代码。

3、系统化完善解决方案

我们在做内存优化的过程中，不仅做了 Android 端的优化工作，还将我们 Android 端一些数据的采集上报到了我们的服务器，然后传到我们的后台，这样，方便我们的无论是 Bug 跟踪人员或者是 Crash 跟踪人员进行一系列问题的解决。

3、如何检测所有不合理的地方？

比如说大图片的检测，我们最初的一个方案是通过继承 ImageView，重写它的 onDraw 方法来实现。但是，我们在推广它的过程中，发现很多开发人员并不接受，因为很多 ImageView 之前已经写过了，你现在让他去替换，工作成本是比较高的。所以说，后来我们就想，有没有一种方案可以免替换，最终我们就找到了 ARTHook 这样一个 Hook 的方案。

如何避免内存抖动？（代码注意事项）

内存抖动是由于短时间内有大量对象进出新生区导致的，它伴随着频繁的 GC，gc 会大量占用 ui 线程和 cpu 资源，会导致 app 整体卡顿。

避免发生内存抖动的几点建议：

- 尽量避免在循环体内创建对象，应该把对象创建移到循环体外。
- 注意自定义 View 的 onDraw()方法会被频繁调用，所以在这里面不应该频繁的创建对象。
- 当需要大量使用 Bitmap 的时候，试着把它们缓存在数组或容器中实现复用。

- 对于能够复用的对象，同理可以使用对象池将它们缓存起来。

需要更全面更深入的理解请查看 Android 性能优化之内存优化、深入探索 Android 内存优化

4、App 绘制优化

1、你在做布局优化的过程中用到了哪些工具？

我在做布局优化的过程中，用到了很多的工具，但是每一个工具都有它不同的使用场景，不同的场景应该使用不同的工具。下面我从线上和线下两个角度来进行分析。

比如说，我要统计线上的 FPS，我使用的就是 Choreographer 这个类，它具有以下特性：

- 1、能够获取整体的帧率。
- 2、能够带到线上使用。
- 3、它获取的帧率几乎是实时的，能够满足我们的需求。

同时，在线下，如果要去优化布局加载带来的时间消耗，那就需要检测每一个布局的耗时，对此我使用的是 AOP 的方式，它没有侵入性，同时也不需要别的开发同学进行接入，就可以方便地获取每一个布局加载的耗时。如果还要更细粒度地去检测每一个控件的加载耗时，那么就需要使用 LayoutInflaterCompat.setFactory2 这个方法去进行 Hook。

此外，我还使用了 LayoutInspector 和 Systrace 这两个工具，Systrace 可以很方便地看到每帧的具体耗时以及这一帧在布局当中它真正做了什么。而 LayoutInspector 可以很方便地看到每一个界面的布局层级，帮助我们对层级进行优化。

2、布局为什么会导致卡顿，你又是如何优化的？

分析完布局的加载流程之后，我们发现有如下四点可能会导致布局卡顿：

- 1、首先，系统会将我们的 Xml 文件通过 **IO** 的方式映射的方式加载到我们的内存当中，而 IO 的过程可能会导致卡顿。
- 2、其次，布局加载的过程是一个反射的过程，而反射的过程也会可能会导致卡顿。
- 3、同时，这个布局的层级如果比较深，那么进行布局遍历的过程就会比较耗时。
- 4、最后，不合理的嵌套 RelativeLayout 布局也会导致重绘的次数过多。

对此，我们的优化方式有如下几种：

- 1、针对布局加载 Xml 文件的优化，我们使用了异步 Inflate 的方式，即 AsyncLayoutInflater。它的核心原理是在子线程中对我们的 Layout 进行加载，而加载完成之后会将 View 通过 Handler 发送到主线程来使用。所以不会阻塞我们的主线程，加载的时间全部是在异步线程中进行消耗的。而这仅仅是一个从侧面缓解的思路。
- 2、后面，我们发现了一个从根源解决上述痛点的方式，即使用 X2C 框架。它的一个核心原理就是在开发过程我们还是使用的 XML 进行编写布局，但是在编译的时候它会使用 APT 的方式将 XML 布局转换为 Java 的方式进行布局，通过这样的方式去写布局，它有以下优点：1、它省去了使用 IO 的方式去加载 XML 布局的耗时过程。2、它是采用 Java 代码直接 new 的方式去创建控件对象，所以它也没有反射带来的性能损耗。这样就从根本上解决了布局加载过程中带来的问题。
- 3、然后，我们可以使用 ConstraintLayout 去减少我们界面布局的嵌套层级，如果原始布局层级越深，它能减少的层级就越多。而使用它也能避免嵌套 RelativeLayout 布局导致的重绘次数过多。

- 4、最后，我们可以使用 AspectJ 框架（即 AOP）和 LayoutInflaterCompat.setFactory2 的方式分别去建立线下全局的布局加载速度和控件加载速度的监控体系。

3、做完布局优化有哪些成果产出？

- 1、首先，我们建立了一个体系化的监控手段，这里的体系还指的是线上加线下的一个综合方案，针对线下，我们使用 AOP 或者 ARTHook，可以很方便地获取到每一个布局的加载耗时以及每一个控件的加载耗时。针对线上，我们通过 Choreographer.getInstance().postFrameCallback 的方式收集到了 FPS，这样我们可以知道用户在哪些界面出现了丢帧的情况。
- 2、然后，对于布局监控方面，我们设立了 FPS、布局加载时间、布局层级等一系列指标。
- 3、最后，在每一个版本上线之前，我们都会对我们的核心路径进行一次 Review，确保我们的 FPS、布局加载时间、布局层级等达到一个合理的状态。

4、你是怎么做卡顿优化的？

从项目的初期到壮大期，最后再到成熟期，每一个阶段都针对卡顿优化做了不同的处理。各个阶段所做的事情如下所示：

- 1、系统工具定位、解决
- 2、自动化卡顿方案及优化
- 3、线上监控及线下监测工具的建设

我做卡顿优化也是经历了一些阶段，最初我们的项目当中的一些模块出现了卡顿之后，我是通过系统工具进行了定位，我使用了 Systrace，然后看了卡顿周期内的 CPU 状况，同时结合代码，对这个模块进行了重构，将部分代码进行了异步和延迟，在项目初期就是这样解决了问题。但是呢，随着我们项目的扩大，线下

卡顿的问题也越来越多，同时，在线上，也有卡顿的反馈，但是线上的反馈卡顿，我们在线下难以复现，于是我们开始寻找自动化的卡顿监测方案，其思路是来自于 Android 的消息处理机制，主线程执行任何代码都会回到 Looper.loop 方法当中，而这个方法中有一个 mLogging 对象，它会在每个 message 的执行前后都会被调用，我们就是利用这个前后处理的时机来做到的自动化监测方案的。同时，在这个阶段，我们也完善了线上 ANR 的上报，我们采取的方式就是监控 ANR 的信息，同时结合了 ANR-WatchDog，作为高版本没有文件权限的一个补充方案。在做完这个卡顿检测方案之后呢，我们还做了线上监控及线下检测工具的建设，最终实现了一整套完善，多维度的解决方案。

5、你是怎么样自动化的获取卡顿信息？

我们的思路是来自于 Android 的消息处理机制，主线程执行任何代码它都会走到 Looper.loop 方法当中，而这个函数当中有一个 **mLogging** 对象，它会在每个 message 处理前后都会被调用，而主线程发生了卡顿，那就一定会在 dispatchMessage 方法中执行了耗时的代码，那我们在这个 message 执行之前呢，我们可以在子线程当中去 postDelayed 一个任务，这个 Delayed 的时间就是我们设定的阈值，如果主线程的 messaege 在这个阈值之内完成了，那就取消掉这个子线程当中的任务，如果主线程的 message 在阈值之内没有被完成，那子线程当中的任务就会被执行，它会获取到当前主线程执行的一个堆栈，那我们就可以知道哪里发生了卡顿。

经过实践，我们发现这种方案获取的堆栈信息它不一定是准确的，因为获取到的堆栈信息它很可能是主线程最终执行的一个位置，而真正耗时的地方其实已经执行完成了，于是呢，我们就对这个方案做了一些优化，我们采取了**高频采集**的方案，也就是在一个周期内我们会多次采集主线程的堆栈信息，如果发生了卡顿，那我们就将这些卡顿信息压缩之后上报给 APM 后台，然后找出重复的堆栈信息，这些重复发生的堆栈大概率就是卡顿发生的一个位置，这样就提高了获取卡顿信息的一个准确性。

6、卡顿的一整套解决方案是怎么做的？

首先，针对卡顿，我们采用了**线上、线下工具相结合**的方式，线下工具我们册中医药尽可能早地去暴露问题，而针对于线上工具呢，我们侧重于监控的全面性、自动化以及异常感知的灵敏度。

同时呢，卡顿问题还有很多的难题。比如说**有的代码呢，它不到你卡顿的一个阈值，但是执行过多，或者它错误地执行了很多次，它也会导致用户感官上的一个卡顿**，所以我们在线下通过 AOP 的方式对常见的耗时代码进行了 Hook，然后对一段时间内获取到的数据进行分析，我们就可以知道这些耗时的代码发生的时机和次数以及耗时情况。然后，看它是不是满足我们的一个预期，不满足预期的话，我们就可以直接到线下进行修改。同时，卡顿监控它还有很多容易被忽略的一个**盲区**，比如说生命周期的一个间隔，那对于这种特定的问题呢，我们就采用了编译时注解的方式修改了项目当中所有 Handler 的父类，对于其中的两个方法进行了监控，我们就可以知道主线程 message 的执行时间以及它们的调用堆栈。

对于**线上卡顿**，我们除了计算 App 的卡顿率、ANR 率等常规指标之外呢，我们还计算了页面的秒开率、生命周期的执行时间等等。而且，在卡顿发生的时刻，我们也尽可能多地保存下来了当前的一个场景信息，这为我们之后解决或者复现这个卡顿留下了依据。

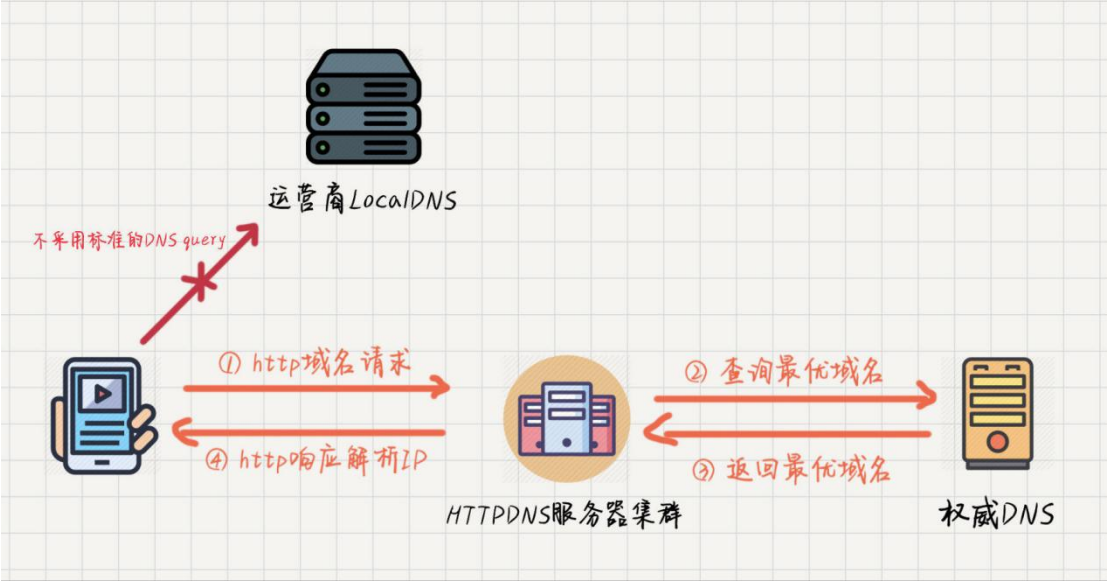
7、TextView setText 耗时的原因，对 TextView 绘制层源码的理解？

8、开放问题：优化一个列表页面的打开速度和流畅性。

需要更全面更深入的理解请查看 Android 性能优化之绘制优化、深入探索 Android 布局优化（上）、深入探索 Android 布局优化（下）

5、App 瘦身

6、网络优化



1、移动端获取网络数据优化的几个点

1、连接复用：节省连接建立时间，如开启 keep-alive。于 Android 来说默认情况下 HttpURLConnection 和 HttpClient 都开启了 keep-alive。只是 2.2 之前 HttpURLConnection 存在影响连接池的 Bug。

2、请求合并：即将多个请求合并为一个进行请求，比较常见的就是网页中的 CSS Image Sprites。如果某个页面内请求过多，也可以考虑做一定的请求合并。

3、减少请求数据的大小：对于 post 请求，body 可以做 gzip 压缩的，header 也可以做数据压缩(不过只支持 http 2.0)。返回数据的 body 也可以做 gzip 压缩，body 数据体积可以缩小到原来的 30%左右（也可以考虑压缩返回的 json 数据的 key 数据的体积，尤其是针对返回数据格式变化不大的情况，支付宝聊天返回的数据用到了）。

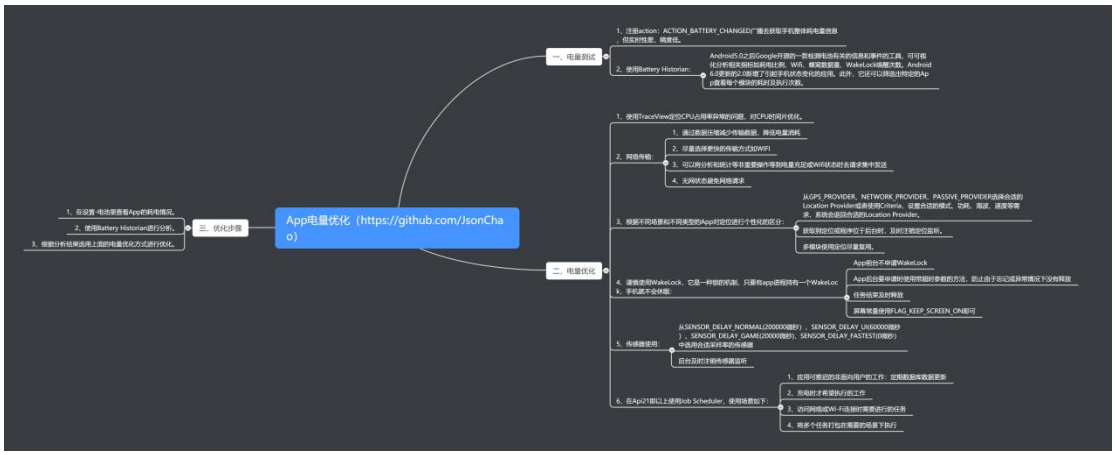
4、根据用户的当前的网络质量来判断下载什么质量的图片（电商用的比较多）。

5、使用 HttpDNS 优化 DNS：DNS 存在解析慢和 DNS 劫持等问题，DNS 不仅支持 UDP，它还支持 TCP，但是大部分标准的 DNS 都是基于 UDP 与 DNS 服务器的 53 端口进行交互。HTTPDNS 则不同，顾名思义它是利用 HTTP 协议与 DNS 服务器的 80 端口进行交互。不走传统的 DNS 解析，从而绕过运营商的 LocalDNS 服务器，有效的防止了域名劫持，提高域名解析的效率。

2、客户端网络安全实现

3、设计一个网络优化方案，针对移动端弱网环境。

7、App 电量优化



8、安卓的安全优化

1、提高 app 安全性的方法？

2、安卓的 app 加固如何做？

3、安卓的混淆原理是什么？

4、谈谈你对安卓签名的理解。

9、为什么 WebView 加载会慢呢？

这是因为在客户端中，加载 H5 页面之前，需要先初始化 WebView，在 WebView 完全初始化完成之前，后续的界面加载过程都是被阻塞的。

优化手段围绕着以下两个点进行：

- 预加载 WebView。
- 加载 WebView 的同时，请求 H5 页面数据。

因此常见的方法是：

- 全局 WebView。
- 客户端代理页面请求。WebView 初始化完成后向客户端请求数据。
- asset 存放离线包。

除此之外还有一些其他的优化手段：

- 脚本执行慢，可以让脚本最后运行，不阻塞页面解析。
- DNS 链接慢，可以让客户端复用使用的域名与链接。
- React 框架代码执行慢，可以将这部分代码拆分出来，提前进行解析。

10、如何优化自定义 View

为了加速你的 view，对于频繁调用的方法，需要尽量减少不必要的代码。先从 onDraw 开始，需要特别注意不应该在这里做内存分配的事情，因为它会导致 GC，从而导致卡顿。在初始化或者动画间隙期间做分配内存的动作。不要在动画正在执行的时候做内存分配的事情。

你还需要尽可能的减少 onDraw 被调用的次数，大多数时候导致 onDraw 都是因为调用了 invalidate().因此请尽量减少调用 invaildate()的次数。如果可能的话，

尽量调用含有 4 个参数的 invalidate()方法而不是没有参数的 invalidate()。没有参数的 invalidate 会强制重绘整个 view。

另外一个非常耗时的操作是请求 layout。任何时候执行 requestLayout()，会使得 Android UI 系统去遍历整个 View 的层级来计算出每一个 view 的大小。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持 View 的层级是扁平化的，这样对提高效率很有帮助。

如果你有一个复杂的 UI，你应该考虑写一个自定义的 ViewGroup 来执行他的 layout 操作。与内置的 view 不同，自定义的 view 可以使得程序仅仅测量这一部分，这避免了遍历整个 view 的层级结构来计算大小。

11、FC(Force Close)什么时候会出现？

Error、OOM，StackOverFlowError、Runtime,比如说空指针异常

解决的办法：

- 注意内存的使用和管理
- 使用 Thread.UncaughtExceptionHandler 接口

12、Java 多线程引发的性能问题，怎么解决？

13、TraceView 的实现原理，分析数据误差来源。

14、是否使用过 SysTrace，原理的了解？

15、mmap + native 日志优化？

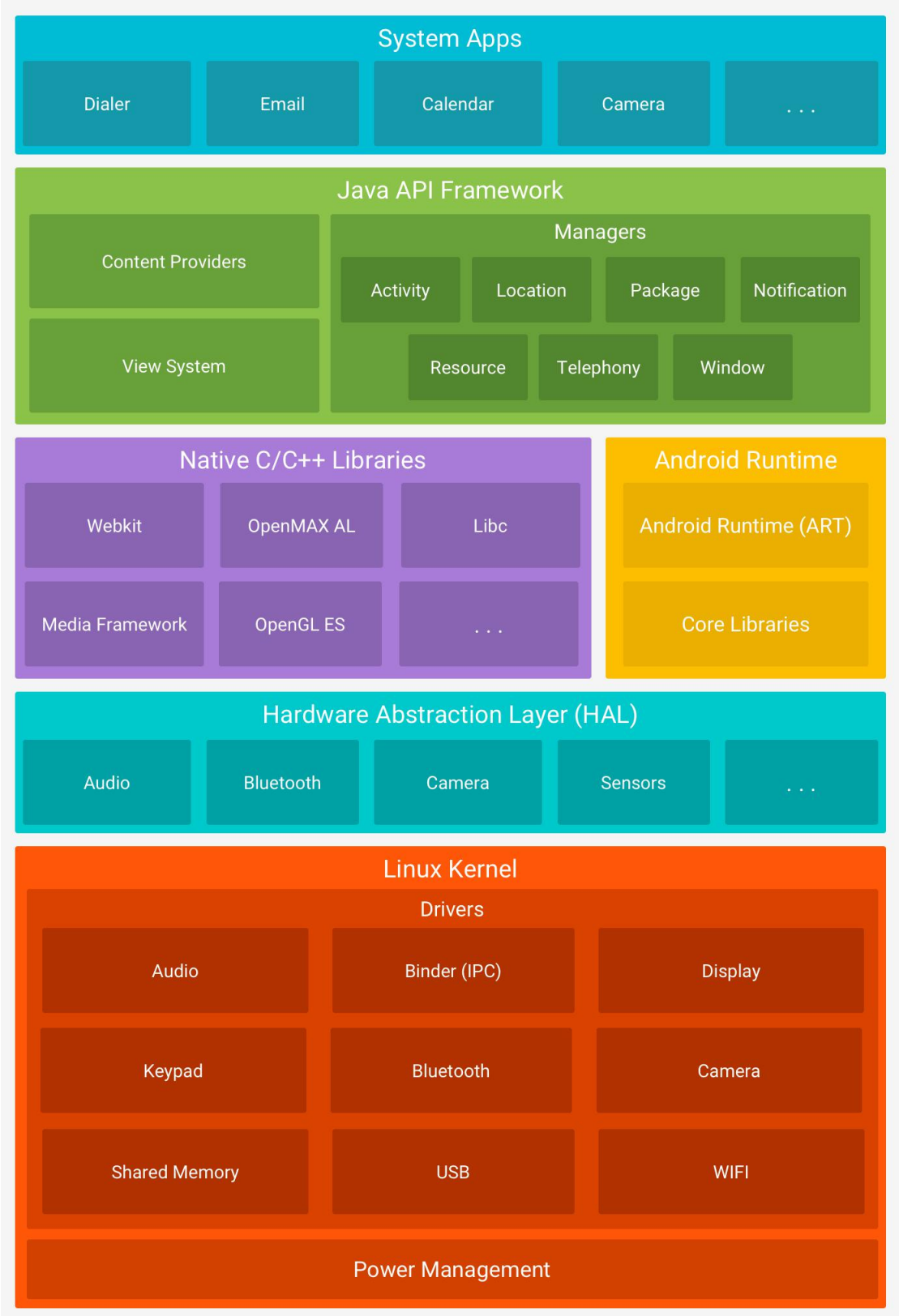
传统日志打印有两个性能问题，一个是反复操作文件描述符表，一个是反复进入内核态。所以需要使用 mmap 的方式去直接读写内存。

二、Android Framework 相关

1、Android 系统架构

Android 是一种基于 Linux 的开放源代码软件栈，为广泛的设备和机型而创建。

下图所示为 Android 平台的五大组件：



1.应用程序

Android 随附一套用于电子邮件、短信、日历、互联网浏览和联系人等的核心应用。平台随附的应用与用户可以选择安装的应用一样，没有特殊状态。因此第三方应用可成为用户的默认网络浏览器、短信 Messenger 甚至默认键盘（有一些例外，例如系统的“设置”应用）。

系统应用可用作用户的应用，以及提供开发者可从其自己的应用访问的主要功能。例如，如果您的应用要发短信，您无需自己构建该功能，可以改为调用已安装的短信应用向您指定的接收者发送消息。

2、Java API 框架

您可通过以 Java 语言编写的 API 使用 Android OS 的整个功能集。这些 API 形成创建 Android 应用所需的构建块，它们可简化核心模块化系统组件和服务的重复使用，包括以下组件和服务：

- 丰富、可扩展的视图系统，可用以构建应用的 UI，包括列表、网格、文本框、按钮甚至可嵌入的网络浏览器
- 资源管理器，用于访问非代码资源，例如本地化的字符串、图形和布局文件
- 通知管理器，可让所有应用在状态栏中显示自定义提醒
- Activity 管理器，用于管理应用的生命周期，提供常见的导航返回栈
- 内容提供程序，可让应用访问其他应用（例如“联系人”应用）中的数据或者共享其自己的数据

开发者可以完全访问 Android 系统应用使用的框架 API。

3、系统运行库

1)原生 C/C++ 库

许多核心 Android 系统组件和服务（例如 ART 和 HAL）构建自原生代码，需要以 C 和 C++ 编写的原生库。Android 平台提供 Java 框架 API 以向应用显示其中部分原生库的功能。例如，您可以通过 Android 框架的 Java OpenGL API 访问 OpenGL ES，以支持在应用中绘制和操作 2D 和 3D 图形。如果开发的是需要 C 或 C++ 代码的应用，可以使用 Android NDK 直接从原生代码访问某些原生平台库。

2)Android Runtime

对于运行 Android 5.0（API 级别 21）或更高版本的设备，每个应用都在其自己的进程中运行，并且有其自己的 Android Runtime (ART) 实例。ART 编写为通过执行 DEX 文件在低内存设备上运行多个虚拟机，DEX 文件是一种专为 Android 设计的字节码格式，经过优化，使用的内存很少。编译工具链（例如 Jack）将 Java 源代码编译为 DEX 字节码，使其可在 Android 平台上运行。

ART 的部分主要功能包括：

- 预先 (AOT) 和即时 (JIT) 编译
- 优化的垃圾回收 (GC)
- 更好的调试支持，包括专用采样分析器、详细的诊断异常和崩溃报告，并且能够设置监视点以监控特定字段

在 Android 版本 5.0（API 级别 21）之前，Dalvik 是 Android Runtime。如果您的应用在 ART 上运行效果很好，那么它应该也可在 Dalvik 上运行，但反过来不一定。

Android 还包含一套核心运行时库，可提供 Java API 框架使用的 Java 编程语言大部分功能，包括一些 Java 8 语言功能。

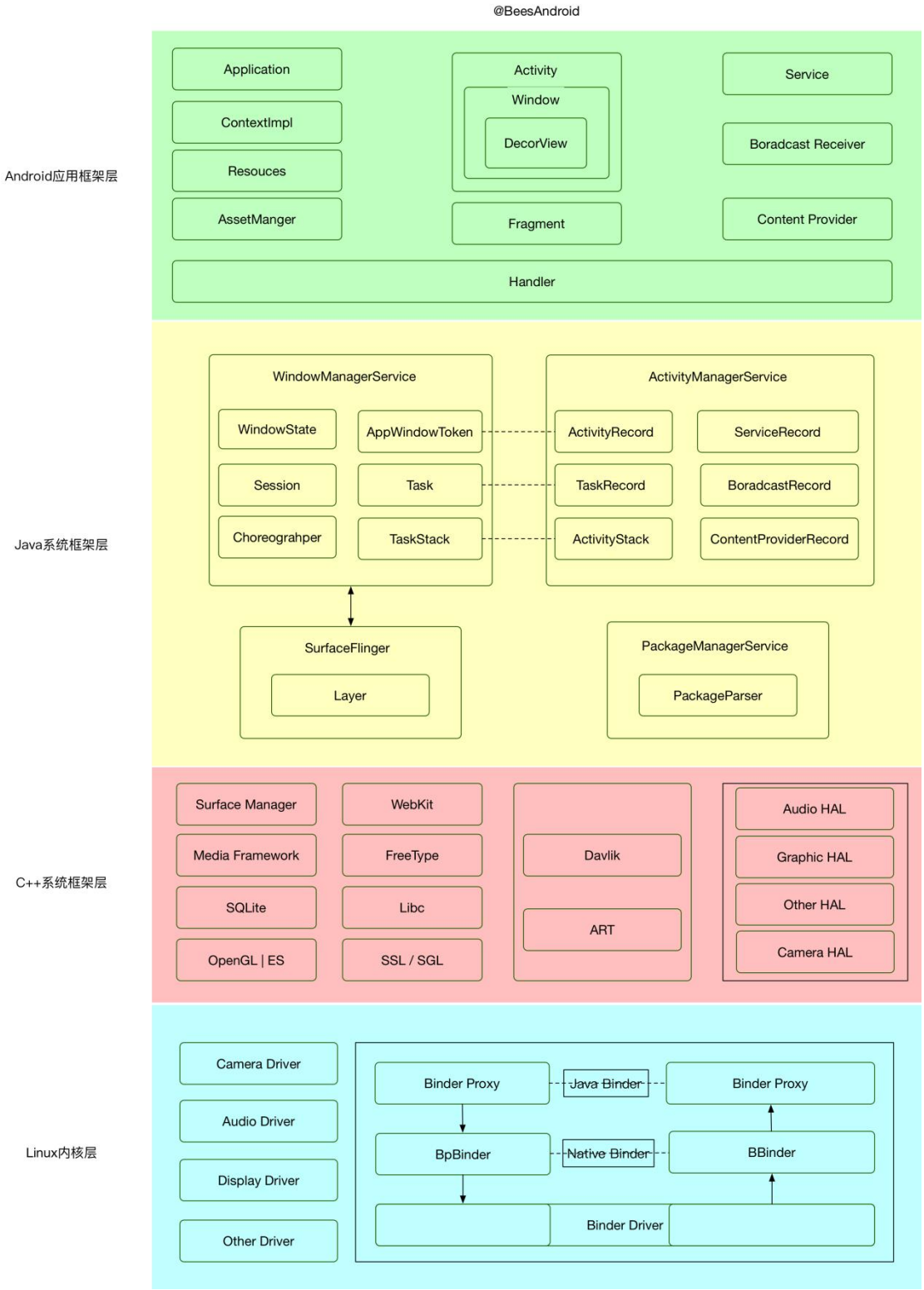
4、硬件抽象层 (HAL)

硬件抽象层 (HAL) 提供标准界面，向更高级别的 Java API 框架显示设备硬件功能。HAL 包含多个库模块，其中每个模块都为特定类型的硬件组件实现一个界面，例如相机或蓝牙模块。当框架 API 要求访问设备硬件时，Android 系统将为该硬件组件加载库模块。

5、Linux 内核

Android 平台的基础是 Linux 内核。例如，Android Runtime (ART) 依靠 Linux 内核来执行底层功能，例如线程和低层内存管理。使用 Linux 内核可让 Android 利用主要安全功能，并且允许设备制造商为著名的内核开发硬件驱动程序。

对于 Android 应用开发来说，最好能手绘下面的系统架构图：



2、View 的事件分发机制？滑动冲突怎么解决？

了解 Activity 的构成

一个 Activity 包含了一个 Window 对象，这个对象是由 PhoneWindow 来实现的。PhoneWindow 将 DecorView 作为整个应用窗口的根 View，而这个 DecorView 又将屏幕划分为两个区域：一个是 TitleView，另一个是 ContentView，而我们平时所写的就是展示在 ContentView 中的。

触摸事件的类型

触摸事件对应的是 MotionEvent 类，事件的类型主要有如下三种：

- ACTION_DOWN
- ACTION_MOVE(移动的距离超过一定的阈值会被判定为 ACTION_MOVE 操作)
- ACTION_UP

View 事件分发本质就是对 MotionEvent 事件分发的过程。即当一个 MotionEvent 发生后，系统将这个点击事件传递到一个具体的 View 上。

事件分发流程

事件分发过程由三个方法共同完成：

dispatchTouchEvent：方法返回值为 true 表示事件被当前视图消费掉；返回为 super.dispatchTouchEvent 表示继续分发该事件，返回为 false 表示交给父类的 onTouchEvent 处理。

onInterceptTouchEvent：方法返回值为 true 表示拦截这个事件并交由自身的 onTouchEvent 方法进行消费；返回 false 表示不拦截，需要继续传递给子视图。如果 return super.onInterceptTouchEvent(ev)， 事件拦截分两种情况:

- 1.如果该View存在子View且点击到了该子View，则不拦截，继续分发 给子 View 处理，此时相当于 return false。
- 2.如果该 View 没有子 View 或者有子 View 但是没有点击中子 View(此时 ViewGroup 相当于普通 View)，则交由该 View 的 onTouchEvent 响应，此时相当于 return true。

注意：一般的 LinearLayout、 RelativeLayout、FrameLayout 等 ViewGroup 默认不拦截， 而 ScrollView、ListView 等 ViewGroup 则可能拦截，得看具体情况。

onTouchEvent：方法返回值为 true 表示当前视图可以处理对应的事件；返回值为 false 表示当前视图不处理这个事件，它会被传递给父视图的 onTouchEvent 方法进行处理。如果 return super.onTouchEvent(ev)，事件处理分为两种情况：

- 1.如果该 View 是 clickable 或者 longclickable 的,则会返回 true，表示消费了该事件，与返回 true 一样;
- 2.如果该 View 不是 clickable 或者 longclickable 的,则会返回 false，表示不消费该事件,将会向上传递,与返回 false 一样。

注意：在 Android 系统中，拥有事件传递处理能力的类有以下三种：

- Activity：拥有分发和消费两个方法。
- ViewGroup：拥有分发、拦截和消费三个方法。
- View：拥有分发、消费两个方法。

三个方法的关系用伪代码表示如下：

```
public boolean dispatchTouchEvent(MotionEvent ev) {
```

```
    boolean consume = false;

    if (onInterceptTouchEvent(ev)) {

        consume = onTouchEvent(ev);

    } else {

        coonsume = child.dispatchTouchEvent(ev);

    }



    return consume;

}
```

通过上面的伪代码，我们可以大致了解点击事件的传递规则：对应一个根 ViewGroup 来说，点击事件产生后，首先会传递给它，这是它的 dispatchTouchEvent 就会被调用，如果这个 ViewGroup 的 onInterceptTouchEvent 方法返回 true 就表示它要拦截当前事件，接着事件就会交给这个 ViewGroup 处理，这时如果它的 mOnTouchListener 被设置，则 onTouch 会被调用，否则 onTouchEvent 会被调用。在 onTouchEvent 中，如果设置了 mOnCLickListener，则 onClick 会被调用。只要 View 的 CLICKABLE 和 LONG_CLICKABLE 有一个为 true，onTouchEvent()就会返回 true 消耗这个事件。

如果这个 ViewGroup 的 onInterceptTouchEvent 方法返回 false 就表示它不拦截当前事件，这时当前事件就会继续传递给它的子元素，接着子元素的 dispatchTouchEvent 方法就会被调用，如此反复直到事件被最终处理。

一些重要的结论：

1、事件传递优先级：onTouchListener.onTouch > onTouchEvent > onClickListener.onClick。

2、正常情况下，一个时间序列只能被一个 View 拦截且消耗。因为一旦一个元素拦截了此事件，那么同一个事件序列内的所有事件都会直接交给它处理（即不会再调用这个 View 的拦截方法去询问它是否要拦截了，而是把剩余的 ACTION_MOVE、ACTION_DOWN 等事件直接交给它来处理）。特例：通过将重写 View 的 onTouchEvent 返回 false 可强行将事件转交给其他 View 处理。

3、如果 View 不消耗除 ACTION_DOWN 以外的其他事件，那么这个点击事件会消失，此时父元素的 onTouchEvent 并不会被调用，并且当前 View 可以持续收到后续的事件，最终这些消失的点击事件会传递给 Activity 处理。

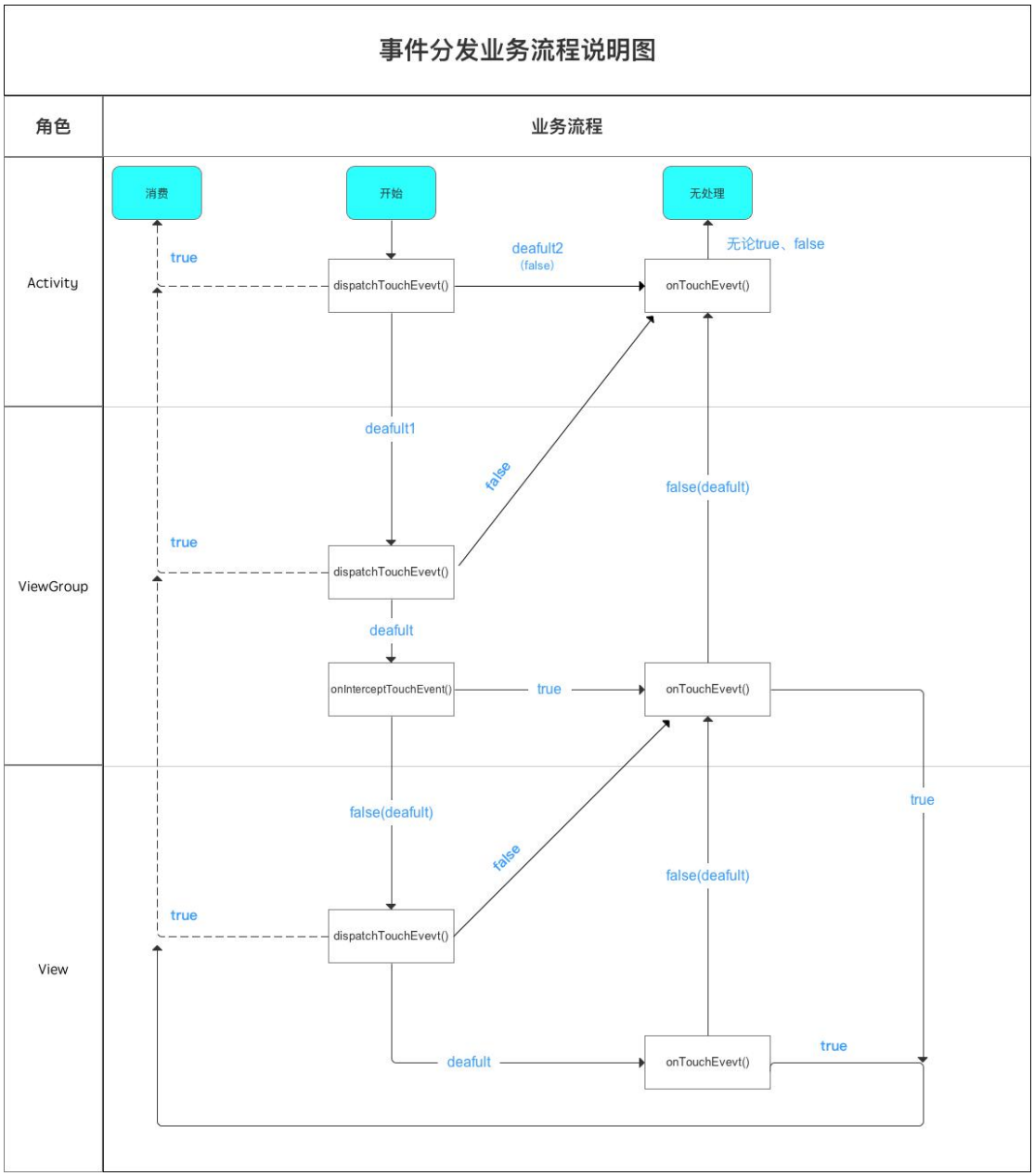
4、ViewGroup 默认不拦截任何事件（返回 false）。

5、View 的 onTouchEvent 默认都会消耗事件（返回 true），除非它是不可点击的（clickable 和 longClickable 同时为 false）。View 的 longClickable 属性默认都为 false，clickable 属性要分情况，比如 Button 的 clickable 属性默认为 true，而 TextView 的 clickable 默认为 false。

6、View 的 enable 属性不影响 onTouchEvent 的默认返回值。

7、通过 requestDisallowInterceptTouchEvent 方法可以在子元素中干预父元素的事件分发过程，但是 ACTION_DOWN 事件除外。

记住这个图的传递顺序,面试的时候能够画出来,就很详细了：



ACTION_CANCEL 什么时候触发，触摸 button 然后滑动到外部抬起会触发点击事件吗，再滑动回去抬起会么？

- 一般 ACTION_CANCEL 和 ACTION_UP 都作为 View 一段事件处理的结束。

  如果在父 View 中拦截 ACTION_UP 或 ACTION_MOVE，在第一次父视图拦截消息的瞬间，父视图指定子视图不接受后续消息了，同时子视图会收到 ACTION_CANCEL 事件。

- 如果触摸某个控件，但是又不是在这个控件的区域上抬起（移动到别的地方了），就会出现 action_cancel。

点击事件被拦截，但是想传到下面的 View，如何操作？

重写子类的 requestDisallowInterceptTouchEvent()方法返回 true 就不会执行父类的 onInterceptTouchEvent()，即可将点击事件传到下面的 View。

如何解决 View 的事件冲突？举个开发中遇到的例子？

常见开发中事件冲突的有 ScrollView 与 RecyclerView 的滑动冲突、RecyclerView 内嵌同时滑动同一方向。

滑动冲突的处理规则：

- 对于由于外部滑动和内部滑动方向不一致导致的滑动冲突，可以根据滑动的方向判断谁来拦截事件。
- 对于由于外部滑动方向和内部滑动方向一致导致的滑动冲突，可以根据业务需求，规定何时让外部 View 拦截事件，何时由内部 View 拦截事件。
- 对于上面两种情况的嵌套，相对复杂，可同样根据需求在业务上找到突破点。

滑动冲突的实现方法：

- 外部拦截法：指点击事件都先经过父容器的拦截处理，如果父容器需要此事件就拦截，否则就不拦截。具体方法：需要重写父容器的 onInterceptTouchEvent 方法，在内部做出相应的拦截。
- 内部拦截法：指父容器不拦截任何事件，而将所有的事件都传递给子容器，如果子容器需要此事件就直接消耗，否则就交由父容器进行处理。具体方法：需要配合 requestDisallowInterceptTouchEvent 方法。

加深理解，GOGOGO

3、View 的绘制流程？

DecorView 被加载到 Window 中

- 从 Activity 的 startActivity 开始，最终调用到 ActivityThread 的 handleLaunchActivity 方法来创建 Activity，首先，会调用 performLaunchActivity 方法，内部会执行 Activity 的 onCreate 方法，从而完成 DecorView 和 Activity 的创建。然后，会调用 handleResumeActivity，里面首先会调用 performResumeActivity 去执行 Activity 的 onResume()方法，执行完后会得到一个 ActivityClientRecord 对象，然后通过 r.window.getDecorView()的方式得到 DecorView，然后会通过 a.getWindowManager() 得到 WindowManager，最终调用其 addView()方法将 DecorView 加进去。
- WindowManager 的实现类是 WindowManagerImpl，它内部会将 addView 的逻辑委托给 WindowManagerGlobal，可见这里使用了接口隔离和委托模式将实现和抽象充分解耦。在 WindowManagerGlobal 的 addView()方法中不仅会将 DecorView 添加到 Window 中，同时会创建 ViewRootImpl 对象，并将 ViewRootImpl 对象和 DecorView 通过 root.setView()把 DecorView 加载到 Window 中。这里的 ViewRootImpl 是 ViewRoot 的实现类，是连接 WindowManager 和 DecorView 的纽带。View 的三大流程均是通过 ViewRoot 来完成的。

了解绘制的整体流程

绘制会从根视图 ViewRoot 的 performTraversals()方法开始，从上到下遍历整个视图树，每个 View 控件负责绘制自己，而 ViewGroup 还需要负责通知自己的子 View 进行绘制操作。

理解 MeasureSpec

MeasureSpec 表示的是一个 32 位的整形值，它的高 2 位表示测量模式 SpecMode，低 30 位表示某种测量模式下的规格大小 SpecSize。MeasureSpec 是 View 类的一个静态内部类，用来说明应该如何测量这个 View。它由三种测量模式，如下：

- EXACTLY：精确测量模式，视图宽高指定为 match_parent 或具体数值时生效，表示父视图已经决定了子视图的精确大小，这种模式下 View 的测量值就是 SpecSize 的值。
- AT_MOST：最大值测量模式，当视图的宽高指定为 wrap_content 时生效，此时子视图的尺寸可以是不超过父视图允许的最大尺寸的任何尺寸。
- UNSPECIFIED：不指定测量模式，父视图没有限制子视图的大小，子视图可以是想要的任何尺寸，通常用于系统内部，应用开发中很少用到。

MeasureSpec 通过将 SpecMode 和 SpecSize 打包成一个 int 值来避免过多的对象内存分配，为了方便操作，其提供了打包和解包的方法，打包方法为 makeMeasureSpec，解包方法为 getMode 和 getSize。

普通 View 的 MeasureSpec 的创建规则如下：

对于 DecorView 而言，它的 MeasureSpec 由窗口尺寸和其自身的 LayoutParams 共同决定；对于普通的 View，它的 MeasureSpec 由父视图的 MeasureSpec 和其自身的 LayoutParams 共同决定。

如何根据 MeasureSpec 去实现一个瀑布流的自定义 ViewGroup？

View 绘制流程之 Measure

- 首先，在 ViewGroup 中的 measureChildren() 方法中会遍历测量 ViewGroup 中所有的 View，当 View 的可见性处于 GONE 状态时，不对其进行测量。
- 然后，测量某个指定的 View 时，根据父容器的 MeasureSpec 和子 View 的 LayoutParams 等信息计算子 View 的 MeasureSpec。
- 最后，将计算出的 MeasureSpec 传入 View 的 measure 方法，这里 ViewGroup 没有定义测量的具体过程，因为 ViewGroup 是一个抽象类，其测量过程的 onMeasure 方法需要各个子类去实现。不同的 ViewGroup 子类有不同的布局特性，这导致它们的测量细节各不相同，如果需要自定义测量过程，则子类可以重写这个方法。（setMeasureDimension 方法用于设置 View 的测量宽高，如果 View 没有重写 onMeasure 方法，则会默认调用 getDefaultSize 来获得 View 的宽高）

getSuggestMinimumWidth 分析

如果 View 没有设置背景，那么返回 android:minWidth 这个属性所指定的值，这个值可以为 0；如果 View 设置了背景，则返回 android:minWidth 和背景的最小宽度这两者中的最大值。

自定义 View 时手动处理 wrap_content 时的情形

直接继承 View 的控件需要重写 onMeasure 方法并设置 wrap_content 时的自身大小，否则在布局中使用 wrap_content 就相当于使用 match_parent。此时，可以在 wrap_content 的情况下（对应 MeasureSpec.AT_MOST）指定内部宽/高(mWidth 和 mHeight)。

LinearLayout 的 onMeasure 方法实现解析（这里仅分析 measureVertical 核心源码）

系统会遍历子元素并对每个子元素执行 measureChildBeforeLayout 方法，这个方法内部会调用子元素的 measure 方法，这样各个子元素就开始依次进入 measure 过程，并且系统会通过 mTotalLength 这个变量来存储 LinearLayout 在竖直方向的初步高度。每测量一个子元素，mTotalLength 就会增加，增加的部分主要包括了子元素的高度以及子元素在竖直方向上的 margin 等。

在 Activity 中获取某个 View 的宽高

由于View的measure过程和Activity的生命周期方法不是同步执行的，如果View还没有测量完毕，那么获得的宽/高就是 0。所以在 onCreate、onStart、onResume 中均无法正确得到某个 View 的宽高信息。解决方式如下：

- Activity/View#onWindowFocusChanged：此时 View 已经初始化完毕，当 Activity 的窗口得到焦点和失去焦点时均会被调用一次，如果频繁地进行 onResume 和 onPause，那么 onWindowFocusChanged 也会被频繁地调用。
- view.post(runnable)：通过 post 可以将一个 runnable 投递到消息队列的尾部，始化好了然后等待 Looper 调用次 runnable 的时候，View 也已经初始化好了。
- ViewTreeObserver#addOnGlobalLayoutListener：当 View 树的状态发生改变或者 View 树内部的 View 的可见性发生改变时，onGlobalLayout 方法将被回调。
- View.measure(int widthMeasureSpec, int heightMeasureSpec)：match_parent 时不知道 parentSize 的大小，测不出；具体数值时，直接 makeMeasureSpec 固定值，然后调用 view..measure 就可以了；

wrap_content 时，在最大化模式下，用 View 理论上能支持的最大值去构造 MeasureSpec 是合理的。

View 的绘制流程之 Layout

首先，会通过 setFrame 方法来设定 View 的四个顶点的位置，即 View 在父容器中的位置。然后，会执行到 onLayout 空方法，子类如果是 ViewGroup 类型，则重写这个方法，实现 ViewGroup 中所有 View 控件布局流程。

LinearLayout 的 onLayout 方法实现解析（layoutVertical 核心源码）

其中会遍历调用每个子 View 的 setChildFrame 方法为子元素确定对应的位置。其中的 childTop 会逐渐增大，意味着后面的子元素会被放置在靠下的位置。

注意：在 View 的默认实现中，View 的测量宽/高和最终宽/高是相等的，只不过测量宽/高形成于 View 的 measure 过程，而最终宽/高形成于 View 的 layout 过程，即两者的赋值时机不同，测量宽/高的赋值时机稍微早一些。在一些特殊的情况下则两者不相等：

- 重写 View 的 layout 方法,使最终宽度总是比测量宽/高大 100px。
- View 需要多次 measure 才能确定自己的测量宽/高，在前几次测量的过程中，其得出的测量宽/高有可能和最终宽/高不一致，但最终来说，测量宽/高还是和最终宽/高相同。

View 的绘制流程之 Draw

Draw 的基本流程

绘制基本上可以分为六个步骤：

- 首先绘制 View 的背景；
- 如果需要的话，保持 canvas 的图层，为 fading 做准备；
- 然后，绘制 View 的内容；
- 接着，绘制 View 的子 View；
- 如果需要的话，绘制 View 的 fading 边缘并恢复图层；
- 最后，绘制 View 的装饰(例如滚动条等等)。

setWillNotDraw 的作用

如果一个 View 不需要绘制任何内容，那么设置这个标记位为 true 以后，系统会进行相应的优化。

- 默认情况下，View 没有启用这个优化标记位，但是 ViewGroup 会默认启用这个优化标记位。
- 当我们的自定义控件继承于 ViewGroup 并且本身不具备绘制功能时，就可以开启这个标记位从而便于系统进行后续的优化。
- 当明确知道一个 ViewGroup 需要通过 onDraw 来绘制内容时，我们需要显示地关闭 WILL_NOT_DRAW 这个标记位。

Requestlayout，onlayout，onDraw，DrawChild 区别与联系？

requestLayout()方法 ：会导致调用 measure()过程 和 layout()过程，将会根据标志位判断是否需要 ondraw。

onLayout()方法：如果该 View 是 ViewGroup 对象，需要实现该方法，对每个子视图进行布局。

onDraw()方法：绘制视图本身 (每个 View 都需要重载该方法，ViewGroup 不需要实现该方法)。

drawChild()：去重新回调每个子视图的 draw()方法。

invalidate() 和 postInvalidate()的区别 ？

invalidate()与 postInvalidate()都用于刷新 View，主要区别是 invalidate()在主线程中调用，若在子线程中使用需要配合 handler；而 postInvalidate()可在子线程中直接调用。

更详细的内容请点击这里

4、跨进程通信。

Android 中进程和线程的关系？区别？

- 线程是 CPU 调度的最小单元，同时线程是一种有限的系统资源；而进程一般指一个执行单元，在 PC 和移动设备上指一个程序或者一个应用。
- 一般来说，一个 App 程序至少有一个进程，一个进程至少有一个线程（包含与被包含的关系），通俗来讲就是，在 App 这个工厂里面有一个进程，线程就是里面的生产线，但主线程（即主生产线）只有一条，而子线程（即副生产线）可以有多个。
- 进程有自己独立的地址空间，而进程中的线程共享此地址空间，都可以并发执行。

如何开启多进程？应用是否可以开启 N 个进程？

在 AndroidManifest 中给四大组件指定属性 android:process 开启多进程模式，在内存允许的条件下可以开启 N 个进程。

为何需要 IPC？多进程通信可能会出现的问题？

所有运行在不同进程的四大组件（Activity、Service、Receiver、ContentProvider）

共享数据都会失败，这是由于 Android 为每个应用分配了独立的虚拟机，不同的虚拟机在内存分配上有不同的地址空间，这会导致在不同的虚拟机中访问同一个类的对象会产生多份副本。比如常用例子（通过开启多进程获取更大内存空间、两个或者多个应用之间共享数据、微信全家桶）。

一般来说，使用多进程通信会造成如下几方面的问题:

- 静态成员和单例模式完全失效：独立的虚拟机造成。
- 线程同步机制完全失效：独立的虚拟机造成。
- SharedPreferences 的可靠性下降：这是因为 Sp 不支持两个进程并发进行读写，有一定几率导致数据丢失。
- Application 会多次创建：Android 系统在创建新的进程时会分配独立的虚拟机，所以这个过程其实就是启动一个应用的过程，自然也会创建新的 Application。

Android 中 IPC 方式、各种方式优缺点？

进程间通信的方式-对比

名称	优点	缺点	适用场景
Intent	简单易用	只能传输Bundle所支持的数据类型	四大组件间的进程间通信
文件共享	简单易用	不适合高并发	简单的数据共享，无高并发场景
AIDL	功能强大，支持一对多并发实时通信	使用稍微复杂，需要注意线程同步	复杂的进程间调用，Android中最常用
Messenger	比AIDL稍微简单易用些	比AIDL功能弱，只支持一对多串行实时通信	简单的进程间通信
ContentProvider	强大的数据共享能力，可通过call方法扩展	受约束的AIDL，主要对外提供数据线的CRUD操作	进程间的大量数据共享
RemoteViews	在跨进程访问UI方面有奇效	比较小众的通信方式	某些特殊的场景
Socket	跨主机，通信范围广	只能传输原始的字节流	常用于网络通信中

讲讲 AIDL？如何优化多模块都使用 AIDL 的情况？

AIDL(Android Interface Definition Language，Android 接口定义语言)：如果在一个进程中要调用另一个进程中对象的方法，可使用 AIDL 生成可序列化的参数，AIDL 会生成一个服务端对象的代理类，通过它客户端可以实现间接调用服务端对象的方法。

AIDL 的本质是系统提供了一套可快速实现 Binder 的工具。关键类和方法：

- AIDL 接口：继承 IInterface。
- Stub 类：Binder 的实现类，服务端通过这个类来提供服务。
- Proxy 类：服务端的本地代理，客户端通过这个类调用服务端的方法。
- asInterface()：客户端调用，将服务端返回的 Binder 对象，转换成客户端所需要的 AIDL 接口类型的对象。如果客户端和服务端位于同一进程，则直接返回 Stub 对象本身，否则返回系统封装后的 Stub.proxy 对象。
- asBinder()：根据当前调用情况返回代理 Proxy 的 Binder 对象。
- onTransact()：运行在服务端的 Binder 线程池中，当客户端发起跨进程请求时，远程请求会通过系统底层封装后交由此方法来处理。
- transact()：运行在客户端，当客户端发起远程请求的同时将当前线程挂起。之后调用服务端的 onTransact()直到远程请求返回，当前线程才继续执行。

当有多个业务模块都需要 AIDL 来进行 IPC，此时需要为每个模块创建特定的 aidl 文件，那么相应的 Service 就会很多。必然会出现系统资源耗费严重、应用过度重量级的问题。解决办法是建立 Binder 连接池，即将每个业务模块的 Binder 请求统一转发到一个远程 Service 中去执行，从而避免重复创建 Service。

工作原理：每个业务模块创建自己的 AIDL 接口并实现此接口，然后向服务端提供自己的唯一标识和其对应的 Binder 对象。服务端只需要一个 Service 并提供一个 queryBinder 接口，它会根据业务模块的特征来返回相应的 Binder 对象，不同的业务模块拿到所需的 Binder 对象后就可以进行远程方法的调用了。

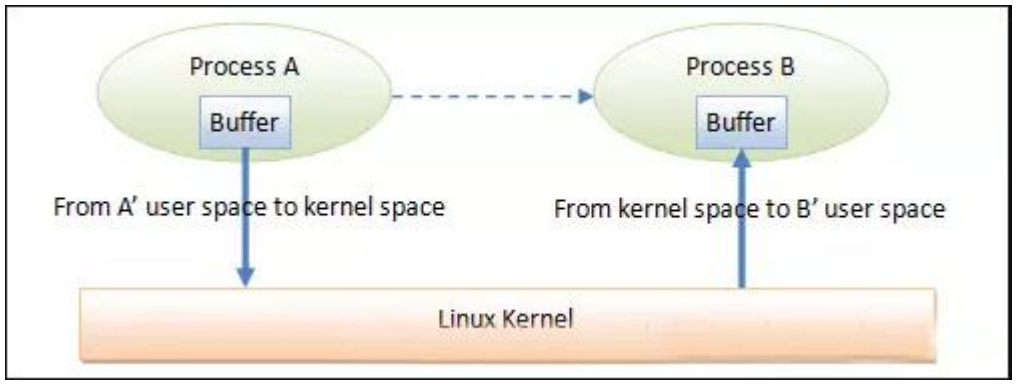
为什么选择 Binder?

为什么选用 Binder，在讨论这个问题之前，我们知道 Android 也是基于 Linux 内核，Linux 现有的进程通信手段有以下几种：

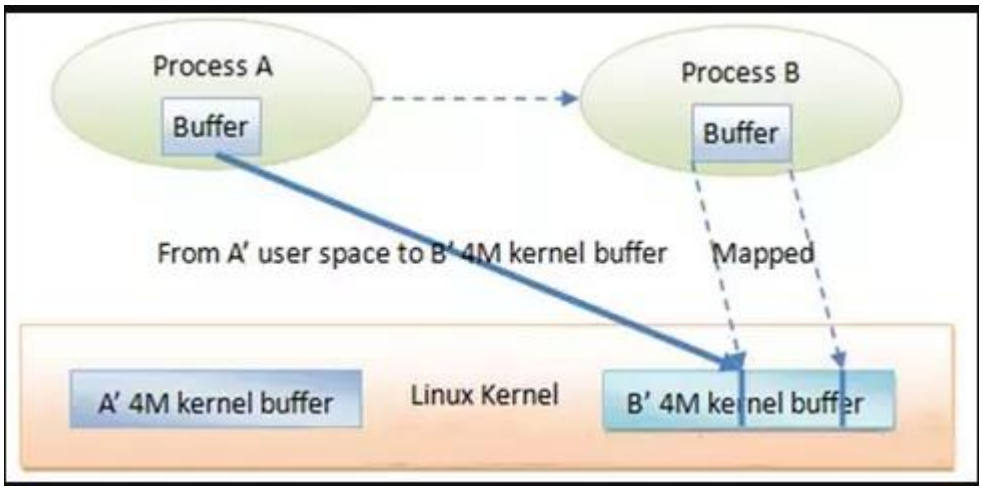
- 管道：在创建时分配一个 page 大小的内存，缓存区大小比较有限；
- 消息队列：信息复制两次，额外的 CPU 消耗；不合适频繁或信息量大的通信；
- 共享内存：无须复制，共享缓冲区直接附加到进程虚拟地址空间，速度快；但进程间的同步问题操作系统无法实现，必须各进程利用同步工具解决；
- 套接字：作为更通用的接口，传输效率低，主要用于不同机器或跨网络的通信；
- 信号量：常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。 不适用于信息交换，更适用于进程中断控制，比如非法内存访问，杀死某个进程等；

既然有现有的 IPC 方式，为什么重新设计一套 Binder 机制呢。主要是出于以上三个方面的考量：

- 1、效率：传输效率主要影响因素是内存拷贝的次数，拷贝次数越少，传输速率越高。从 Android 进程架构角度分析：对于消息队列、Socket 和管道来说，数据先从发送方的缓存区拷贝到内核开辟的缓存区中，再从内核缓存区拷贝到接收方的缓存区，一共两次拷贝，如图：

- 

而对于 Binder 来说，数据从发送方的缓存区拷贝到内核的缓存区，而接收方的缓存区与内核的缓存区是映射到同一块物理地址的，节省了一次数据拷贝的过程，如图：



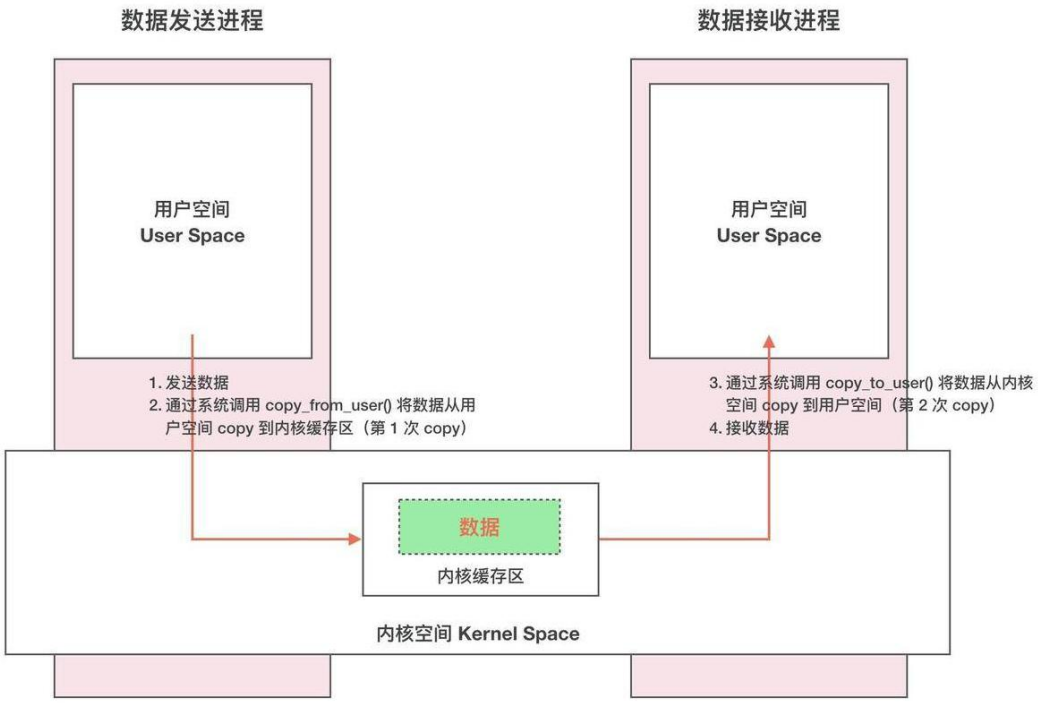
共享内存不需要拷贝，Binder 的性能仅次于共享内存。

- 2、稳定性：上面说到共享内存的性能优于 Binder，那为什么不采用共享内存呢，因为共享内存需要处理并发同步问题，容易出现死锁和资源竞争，稳定性较差。Socket 虽然是基于 C/S 架构的，但是它主要是用于网络间的通信且传输效率较低。Binder 基于 C/S 架构 ，Server 端与 Client 端相对独立，稳定性较好。

- 3、安全性：传统 Linux IPC 的接收方无法获得对方进程可靠的 UID/PID，从而无法鉴别对方身份；而 Binder 机制为每个进程分配了 UID/PID，且在 Binder 通信时会根据 UID/PID 进行有效性检测。

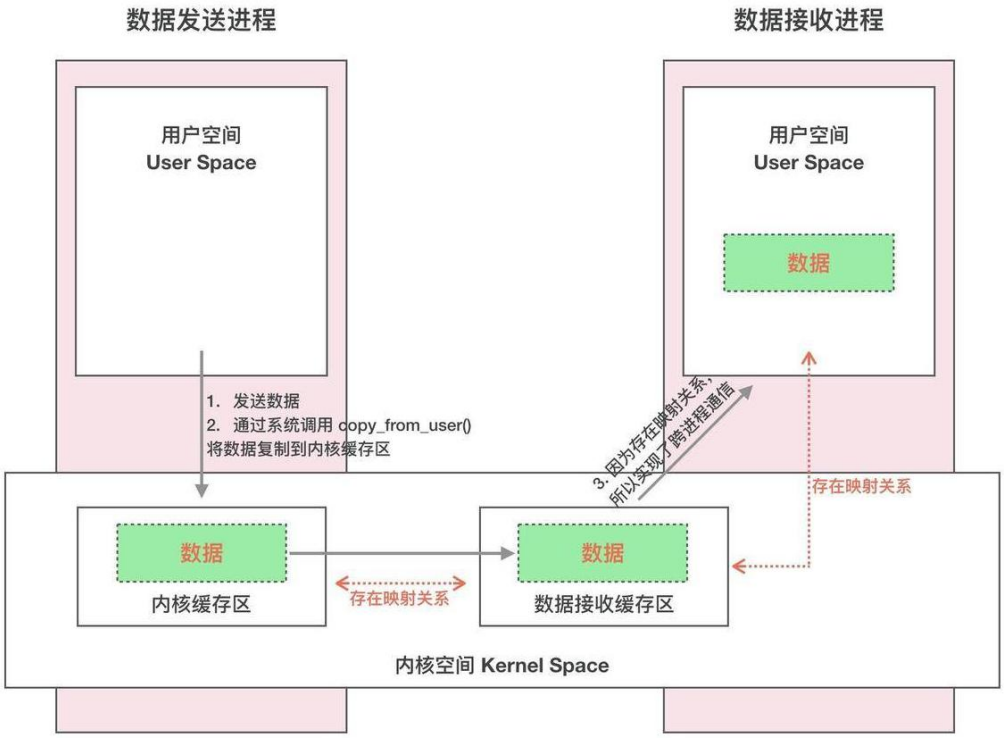
Binder 机制的作用和原理？

Linux 系统将一个进程分为用户空间和内核空间。对于进程之间来说，用户空间的数据不可共享，内核空间的数据可共享，为了保证安全性和独立性，一个进程不能直接操作或者访问另一个进程，即 Android 的进程是相互独立、隔离的，这就需要跨进程之间的数据通信方式。普通的跨进程通信方式一般需要 2 次内存拷贝，如下图所示：



一次完整的 Binder IPC 通信过程通常是这样：

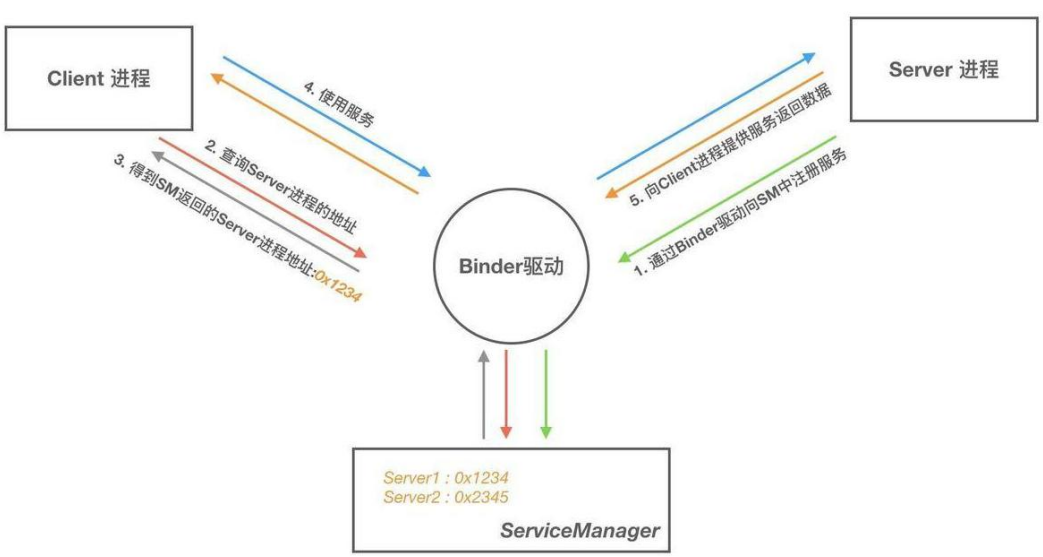
- 首先 Binder 驱动在内核空间创建一个数据接收缓存区。
- 接着在内核空间开辟一块内核缓存区，建立内核缓存区和内核中数据接收缓存区之间的映射关系，以及内核中数据接收缓存区和接收进程用户空间地址的映射关系。
- 发送方进程通过系统调用 copyfromuser() 将数据 copy 到内核中的内核缓存区，由于内核缓存区和接收进程的用户空间存在内存映射，因此也就相当于把数据发送到了接收进程的用户空间，这样便完成了一次进程间的通信。



-

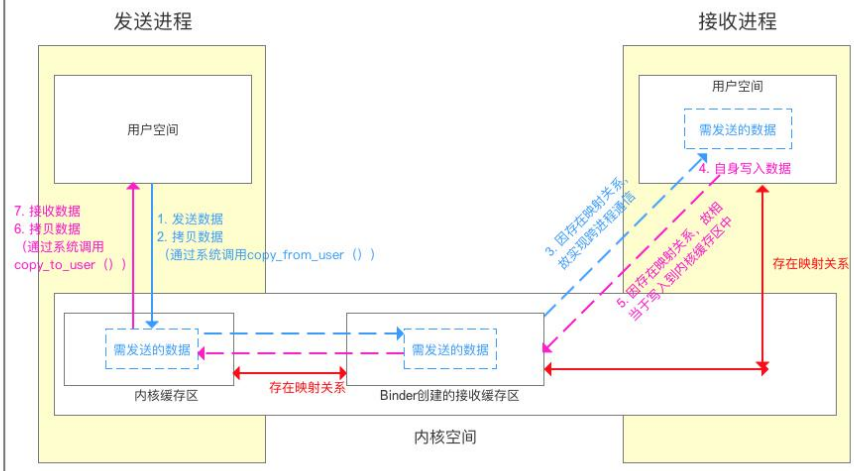
Binder 框架中 ServiceManager 的作用？

Binder 框架 是基于 C/S 架构的。由一系列的组件组成，包括 Client、Server、ServiceManager、Binder 驱动，其中 Client、Server、Service Manager 运行在用户空间，Binder 驱动运行在内核空间。如下图所示：



- Server&Client：服务器&客户端。在 Binder 驱动和 Service Manager 提供的基础设施上，进行 Client-Server 之间的通信。
- ServiceManager（如同 DNS 域名服务器）服务的管理者，将 Binder 名字转换为 Client 中对该 Binder 的引用，使得 Client 可以通过 Binder 名字获得 Server 中 Binder 实体的引用。
- Binder 驱动（如同路由器）：负责进程之间 binder 通信的建立，计数管理以及数据的传递交互等底层支持。

最后，结合 Android 跨进程通信：图文详解 Binder 机制 的总结图来综合理解一下：

<table>
<tr><th>步骤</th><th colspan="3">过程描述</th></tr>
<tr><td>1. 注册服务</td><td colspan="2">1. Server进程 向 Binder驱动 发起服务注册请求
2. Binder驱动 将注册请求转发给Service Manager进程
3. Service Manager进程 添加该Server进程（即已注册服务）</td><td>此时，ServiceManager进程拥有了Server进程的信息</td></tr>
<tr><td>2. 获取服务</td><td colspan="2">1. Client 向 Binder 驱动发起获取服务的请求，传递要获取的服务名称
2. Binder 驱动将该请求转发给 ServiceManager 进程
3. ServiceManager 查找到 Client 需要的 Server 对应的服务信息
4. 通过 Binder 驱动将上述服务信息返回给 Client进程</td><td>此时，Client进程与 Server进程已经建立了连接</td></tr>
<tr><td rowspan="5">3. 使用服务</td><td colspan="2">步骤1： Binder驱动为跨进程通信作准备：实现内存映射
（调用mmap（）系统函数）</td><td>1. Binder驱动 创建一块 接收缓存区
2. 实现地址映射关系：即 根据 ServiceManager进程里的Server信息找到对应的Server 进程
，实现 内核缓存区 和 Server 进程用户空间地址 同时映射到 同1个接收缓存区中
（注：此时仅创建了虚拟区间 & 映射关系，但并无将传输数据）</td></tr>
<tr><td colspan="2">步骤2： Client进程 将参数数据发送到Server进程</td><td>1. Client进程 通过 系统调用copy_from_user（）发送数据到内核空间中的缓存区；（当前线程被挂起）
（ 由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系（同时映射Binder创建的接收缓存区中），故相当于也发送到了Server进程的用户空间地址，即Binder驱动实现了跨进程通信）
3. Binder驱动 通知Server 进程执行 解包</td></tr>
<tr><td colspan="2">步骤3： Server进程 根据Client进程要求 调用目标方法</td><td>1. 收到Binder驱动通知后，Server 进程从线程池中取出 线程，进行数据解包 & 调用目标方法
2. 将最终执行结果写入到自己的共享内存中</td></tr>
<tr><td colspan="2">步骤4：Server进程 将目标方法的结果 返回给Client进程</td><td>1. 将最终执行结果写入存在映射的用户空间的内存区域中
（ 由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系（同时映射Binder创建的接收缓存区中），故相当于也发送到了内核缓存区中）
2. Binder驱动通知Client进程获得返回结果（此时Client进程之前被挂起的线程被重新唤醒）
3. Client进程 通过 系统调用copy_to_user（）从内核缓存区接收Server进程返回的数据</td></tr>
<tr><td>示意图</td><td colspan="2"> 
</td></tr>
<tr><td>优点</td><td colspan="2">• 传输效率高：每次单向通信数据拷贝次数少（1次）、用户空间 & 内核空间可直接通过共享对象直接交互
• 为接收进程 分配了不确定大小的接收缓存区</td></tr>
</table>

Binder 的完整定义

- 从进程间通信的角度看，Binder 是一种进程间通信的机制；
- 从 Server 进程的角度看，Binder 指的是 Server 中的 Binder 实体对象；
- 从 Client 进程的角度看，Binder 指的是 Binder 代理对象，是 Binder 实体对象的一个远程代理;

- 从传输过程的角度看，Binder 是一个可以跨进程传输的对象；Binder 驱动会对这个跨越进程边界的对象对一点点特殊处理，自动完成代理对象和本地对象之间的转换。

手写实现简化版 AMS（AIDL 实现）

与 Binder 相关的几个类的职责:

- IBinder：跨进程通信的 Base 接口，它声明了跨进程通信需要实现的一系列抽象方法，实现了这个接口就说明可以进行跨进程通信，Client 和 Server 都要实现此接口。
- IInterface：这也是一个 Base 接口，用来表示 Server 提供了哪些能力，是 Client 和 Server 通信的协议。
- Binder：提供 Binder 服务的本地对象的基类，它实现了 IBinder 接口，所有本地对象都要继承这个类。
- BinderProxy：在 Binder.java 这个文件中还定义了一个 BinderProxy 类，这个类表示 Binder 代理对象它同样实现了 IBinder 接口，不过它的很多实现都交由 native 层处理。Client 中拿到的实际上是这个代理对象。
- Stub：这个类在编译 aidl 文件后自动生成，它继承自 Binder，表示它是一个 Binder 本地对象；它是一个抽象类，实现了 IInterface 接口，表明它的子类需要实现 Server 将要提供的具体能力(即 aidl 文件中声明的方法)。
- Proxy：它实现了 IInterface 接口，说明它是 Binder 通信过程的一部分；它实现了 aidl 中声明的方法，但最终还是交由其中的 mRemote 成员来处理，说明它是一个代理对象，mRemote 成员实际上就是 BinderProxy。

aidl 文件只是用来定义 C/S 交互的接口，Android 在编译时会自动生成相应的 Java 类，生成的类中包含了 Stub 和 Proxy 静态内部类，用来封装数据转换的过程，实际使用时只关心具体的 Java 接口类即可。为什么 Stub 和 Proxy 是静态内部类呢？这其实只是为了将三个类放在一个文件中，提高代码的聚合性。通过上面的分析，我们其实完全可以不通过 aidl，手动编码来实现 Binder 的通信，下面我们通过编码来实现 ActivityManagerService：

1、首先定义 IActivityManager 接口：

```
public interface IActivityManager extends IInterface {

    //binder 描述符

    String DESCRIPTOR = "android.app.IActivityManager";

    //方法编号

    int TRANSACTION_startActivity = IBinder.FIRST_CALL_TRANSACTION + 0;

    //声明一个启动 activity 的方法，为了简化，这里只传入 intent 参数

    int startActivity(Intent intent) throws RemoteException;

}
```

2、然后，实现 ActivityManagerService 侧的本地 Binder 对象基类：

```
// 名称随意，不一定叫 Stub

public abstract class ActivityManagerNative extends Binder implements
IActivityManager {



    public static IActivityManager asInterface(IBinder obj) {

        if (obj == null) {

            return null;

        }
```

```
        IActivityManager in = (IActivityManager)
obj.queryLocalInterface(IActivityManager.DESCRIPTOR);

        if (in != null) {

            return in;

        }

        //代理对象，见下面的代码

        return new ActivityManagerProxy(obj);

    }



    @Override

    public IBinder asBinder() {

        return this;

    }



    @Override

    protected boolean onTransact(int code, Parcel data, Parcel reply, int flags)
throws RemoteException {

        switch (code) {

            // 获取 binder 描述符

            case INTERFACE_TRANSACTION:

                reply.writeString(IActivityManager.DESCRIPTOR);

                return true;

            // 启动 activity，从 data 中反序列化出 intent 参数后，直接调用子类
startActivity 方法启动 activity。

            case IActivityManager.TRANSACTION_startActivity:

                data.enforceInterface(IActivityManager.DESCRIPTOR);

                Intent intent = Intent.CREATOR.createFromParcel(data);
```

```
                int result = this.startActivity(intent);
                reply.writeNoException();
                reply.writeInt(result);
                return true;
        }
        return super.onTransact(code, data, reply, flags);
    }
}
```

3、接着，实现 Client 侧的代理对象：

```
public class ActivityManagerProxy implements IActivityManager {
    private IBinder mRemote;

    public ActivityManagerProxy(IBinder remote) {
        mRemote = remote;
    }

    @Override
    public IBinder asBinder() {
        return mRemote;
    }

    @Override
    public int startActivity(Intent intent) throws RemoteException {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
```

```
        int result;
        try {
            // 将 intent 参数序列化，写入 data 中
            intent.writeToParcel(data, 0);
            // 调用 BinderProxy 对象的 transact 方法，交由 Binder 驱动处理。
            mRemote.transact(IActivityManager.TRANSACTION_startActivity, data,
reply, 0);
            reply.readException();
            // 等待 server 执行结束后，读取执行结果
            result = reply.readInt();
        } finally {
            data.recycle();
            reply.recycle();
        }
        return result;
    }
}
```

4、最后，实现 Binder 本地对象（IActivityManager 接口）：

```
public class ActivityManagerService extends ActivityManagerNative {
    @Override
    public int startActivity(Intent intent) throws RemoteException {
        // 启动 activity
        return 0;
    }
}
```

简化版的 ActivityManagerService 到这里就已经实现了，剩下就是 Client 只需要获取到 AMS 的代理对象 IActivityManager 就可以通信了。

简单讲讲 binder 驱动吧？

从 Java 层来看就像访问本地接口一样，客户端基于 BinderProxy 服务端基于 IBinder 对象，从 native 层来看来看客户端基于 BpBinder 到 ICPThreadState 到 binder 驱动，服务端由 binder 驱动唤醒 IPCThreadSate 到 BbBinder 。跨进程通信的原理最终是要基于内核的，所以最会会涉及到 binder_open 、binder_mmap 和 binder_ioctl 这三种系统调用。

跨进程传递大内存数据如何做？

binder 肯定是不行的，因为映射的最大内存只有 1M-8K，可以采用 binder + 匿名共享内存的形式，像跨进程传递大的 bitmap 需要打开系统底层的 ashmem 机制。

请按顺序仔细阅读下列文章提升对 Binder 机制的理解程度：

写给 Android 应用工程师的 Binder 原理剖析

Binder 学习指南

Binder 设计与实现

老罗 Binder 机制分析系列或 Android 系统源代码情景分析 Binder 章节

5、Android 系统启动流程是什么？（提示：init 进程 -> Zygote 进程 -> SystemServer 进程 -> 各种系统服务 -> 应用进程）

Android 系统启动的核心流程如下：

- 1、**启动电源以及系统启动**：当电源按下时引导芯片从预定义的地方（固化在 ROM）开始执行，加载引导程序 BootLoader 到 RAM，然后执行。
- 2、**引导程序 BootLoader**：BootLoader 是在 Android 系统开始运行前的一个小程序，主要用于把系统 OS 拉起来并运行。
- 3、**Linux 内核启动**：当内核启动时，设置缓存、被保护存储器、计划列表、加载驱动。当其完成系统设置时，会先在系统文件中寻找 init.rc 文件，并启动 init 进程。
- 4、**init 进程启动**：初始化和启动属性服务，并且启动 Zygote 进程。
- 5、**Zygote 进程启动**：创建 JVM 并为其注册 JNI 方法，创建服务器端 Socket，启动 SystemServer 进程。
- 6、**SystemServer 进程启动**：启动 Binder 线程池和 SystemServiceManager，并且启动各种系统服务。
- 7、**Launcher 启动**：被 SystemServer 进程启动的 AMS 会启动 Launcher，Launcher 启动后会将已安装应用的快捷图标显示到系统桌面上。

需要更详细的分析请查看以下系列文章：

Android 系统启动流程之 init 进程启动

Android 系统启动流程之 Zygote 进程启动

Android 系统启动流程之 SystemServer 进程启动

Android 系统启动流程之 Launcher 进程启动

系统是怎么帮我们启动找到桌面应用的？

通过意图，PMS 会解析所有 apk 的 AndroidManifest.xml ，如果解析过会存到 package.xml 中不会反复解析，PMS 有了它就能找到了。

6、启动一个程序，可以主界面点击图标进入，也可以从一个程序中跳转过去，二者有什么区别？

是因为启动程序（主界面也是一个 app），发现了在这个程序中存在一个设置为的 activity, 所以这个 launcher 会把 icon 提出来，放在主界面上。当用户点击 icon 的时候，发出一个 Intent：

```
Intent intent =
mActivity.getPackageManager().getLaunchIntentForPackage(packageName);

mActivity.startActivity(intent);
```

跳过去可以跳到任意允许的页面，如一个程序可以下载，那么真正下载的页面可能不是首页（也有可能是首页），这时还是构造一个 Intent，startActivity。这个 intent 中的 action 可能有多种 view，download 都有可能。系统会根据第三方程序向系统注册的功能，为你的 Intent 选择可以打开的程序或者页面。所以唯一的一点 不同的是从 icon 的点击启动的 intent 的 action 是相对单一的，从程序中跳转或者启动可能样式更多一些。本质是相同的。

7、AMS 家族重要术语解释。

1.ActivityManagerServices，简称 AMS，服务端对象，负责系统中所有 Activity 的生命周期。

2.ActivityThread，App 的真正入口。当开启 App 之后，调用 main()开始运行，开启消息循环队列，这就是传说的 UI 线程或者叫主线程。与 ActivityManagerService 一起完成 Activity 的管理工作。

3.ApplicationThread，用来实现 ActivityManagerServie 与 ActivityThread 之间的交互。在 ActivityManagerSevice 需要管理相关 Application 中的 Activity 的生命周期时，通过 ApplicationThread 的代理对象与 ActivityThread 通信。

4.ApplicationThreadProxy，是 ApplicationThread 在服务器端的代理，负责和客户端的 ApplicationThread 通信。AMS 就是通过该代理与 ActivityThread 进行通信的。

5.Instrumentation，每一个应用程序只有一个 Instrumetation 对象，每个 Activity 内都有一个对该对象的引用，Instrumentation 可以理解为应用进程的管家，ActivityThread 要创建或暂停某个 Activity 时，都需要通过 Instrumentation 来进行具体的操作。

6.ActivityStack，Activity 在 AMS 的栈管理，用来记录经启动的 Activity 的先后关系，状态信息等。通过 ActivtyStack 决定是否需要启动新的进程。

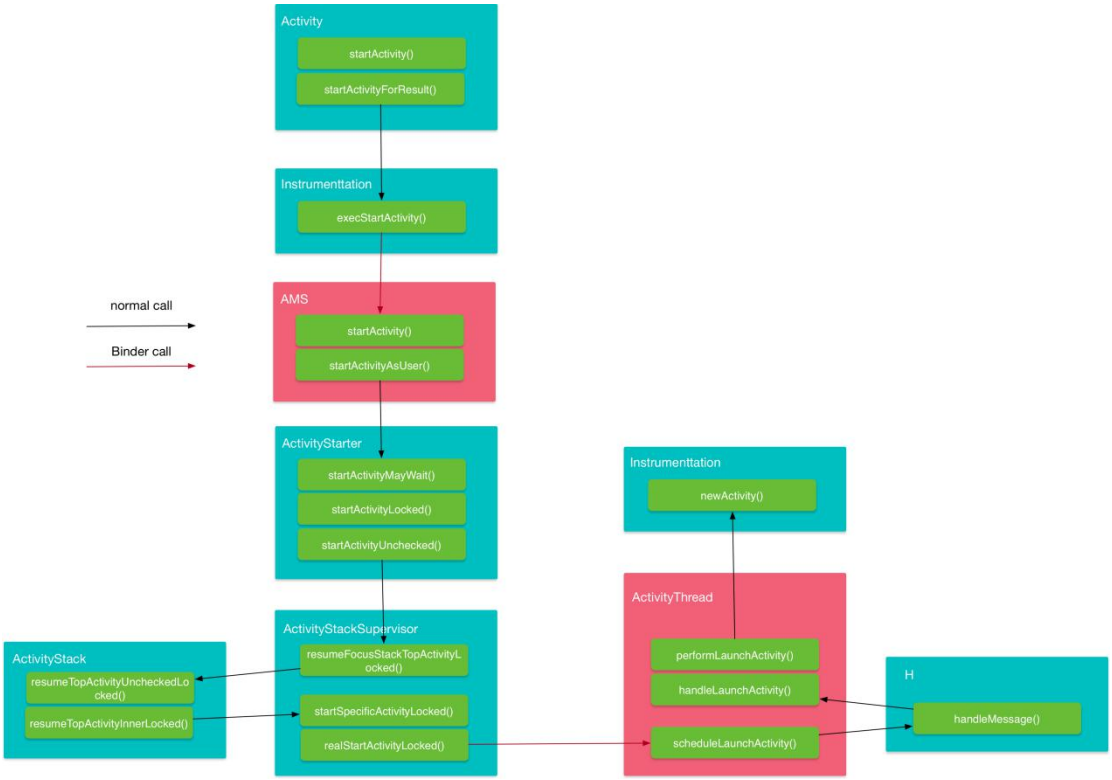
7.ActivityRecord，ActivityStack 的管理对象，每个 Acivity 在 AMS 对应一个 ActivityRecord，来记录 Activity 状态以及其他的管理信息。其实就是服务器端的 Activit 对象的映像。

8.TaskRecord，AMS 抽象出来的一个“任务”的概念，是记录 ActivityRecord 的栈，一个“Task”包含若干个 ActivityRecord。AMS 用 TaskRecord 确保 Activity 启动和退出的顺序。如果你清楚 Activity 的 4 种 launchMode，那么对这概念应该不陌生。

8、App 启动流程（Activity 的冷启动流程）。

点击应用图标后会去启动应用的 Launcher Activity，如果 Launcer Activity 所在的进程没有创建，还会创建新进程，整体的流程就是一个 Activity 的启动流程。

Activity 的启动流程图（放大可查看）如下所示：



整个流程涉及的主要角色有：

- Instrumentation: 监控应用与系统相关的交互行为。
- AMS：组件管理调度中心，什么都不干，但是什么都管。
- ActivityStarter：Activity 启动的控制器，处理 Intent 与 Flag 对 Activity 启动的影响，具体说来有：1 寻找符合启动条件的 Activity，如果有多个，让用户选择；2 校验启动参数的合法性；3 返回 int 参数，代表 Activity 是否启动成功。
- ActivityStackSupervisior：这个类的作用你从它的名字就可以看出来，它用来管理任务栈。

- ActivityStack：用来管理任务栈里的 Activity。
- ActivityThread：最终干活的人，Activity、Service、BroadcastReceiver 的启动、切换、调度等各种操作都在这个类里完成。

注：这里单独提一下 ActivityStackSupervisior，这是高版本才有的类，它用来管理多个 ActivityStack，早期的版本只有一个 ActivityStack 对应着手机屏幕，后来高版本支持多屏以后，就有了多个 ActivityStack，于是就引入了 ActivityStackSupervisior 用来管理多个 ActivityStack。

整个流程主要涉及四个进程：

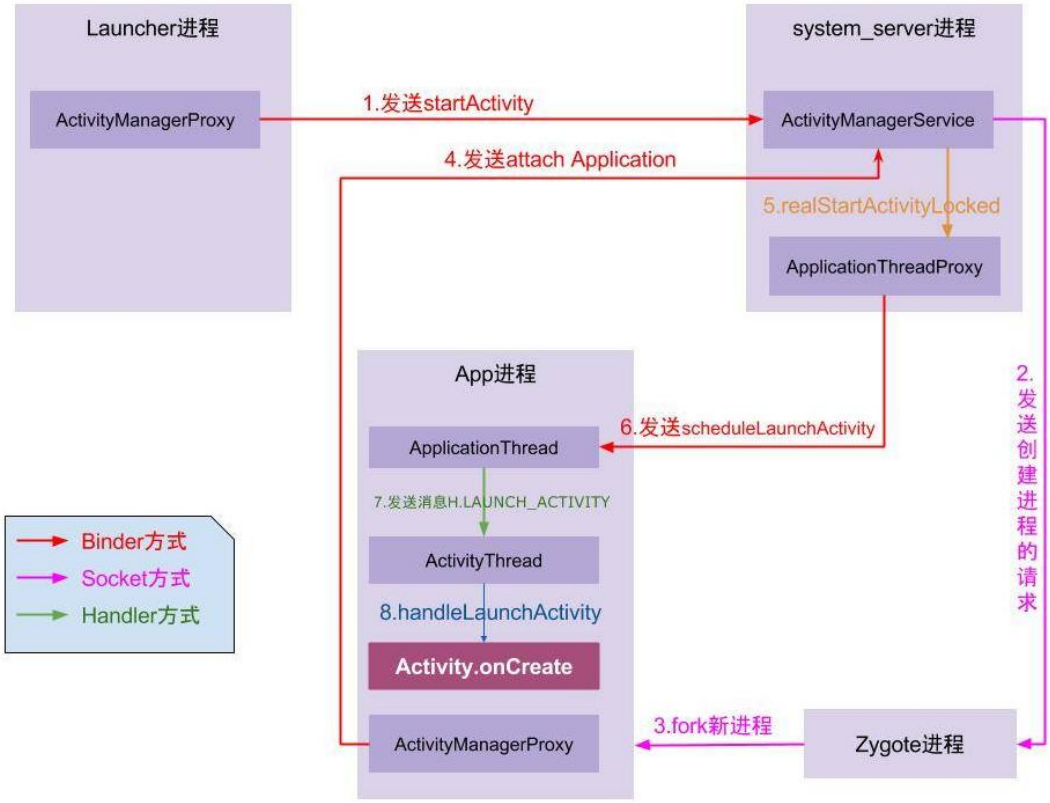
- 调用者进程，如果是在桌面启动应用就是 Launcher 应用进程。
- ActivityManagerService 等待所在的 System Server 进程，该进程主要运行着系统服务组件。
- Zygote 进程，该进程主要用来 fork 新进程。
- 新启动的应用进程，该进程就是用来承载应用运行的进程了，它也是应用的主线程（新创建的进程就是主线程），处理组件生命周期、界面绘制等相关事情。

有了以上的理解，整个流程可以概括如下：

- 1、点击桌面应用图标，Launcher 进程将启动 Activity（MainActivity）的请求以 Binder 的方式发送给了 AMS。
- 2、AMS 接收到启动请求后，交付 ActivityStarter 处理 Intent 和 Flag 等信息，然后再交给 ActivityStackSupervisior/ActivityStack 处理 Activity 进栈相关流程。同时以 Socket 方式请求 Zygote 进程 fork 新进程。
- 3、Zygote 接收到新进程创建请求后 fork 出新进程。

- 4、在新进程里创建 ActivityThread 对象，新创建的进程就是应用的主线程，在主线程里开启 Looper 消息循环，开始处理创建 Activity。
- 5、ActivityThread 利用 ClassLoader 去加载 Activity、创建 Activity 实例，并回调 Activity 的 onCreate()方法，这样便完成了 Activity 的启动。

最后，再看看另一幅启动流程图来加深理解：



9、ActivityThread 工作原理。

10、说下四大组件的启动过程，四大组件的启动与销毁的方式。

广播发送和接收的原理了解吗？

- 继承 BroadcastReceiver，重写 onReceive()方法。
- 通过 Binder 机制向 ActivityManagerService 注册广播。

- 通过 Binder 机制向 ActivityMangerService 发送广播。
- ActivityManagerService 查找符合相应条件的广播（IntentFilter/Permission）的 BroadcastReceiver，将广播发送到 BroadcastReceiver 所在的消息队列中。
- BroadcastReceiver 所在消息队列拿到此广播后，回调它的 onReceive()方法。

11、AMS 是如何管理 Activity 的？

12、理解 Window 和 WindowManager。

1.Window 用于显示 View 和接收各种事件，Window 有三种型：应用 Window(每个 Activity 对应一个 Window)、子 Widow(不能单独存在，附属于特定 Window)、系统 window(toast 和状态栏)

2.Window 分层级，应用 Window 在 1-99、子 Window 在 1000-1999、系统 Window 在 2000-2999.WindowManager 提供了增改 View 的三个功能。

3.Window 是个抽象概念：每一个 Window 对应着一个 ViewRootImpl，Window 通过 ViewRootImpl 来和 View 建立联系，View 是 Window 存在的实体，只能通过 WindowManager 来访问 Window。

4.WindowManager 的实现是 WindowManagerImpl，其再委托 WindowManagerGlobal 来对 Window 进行操作，其中有四种 List 分别储存对应的 View、ViewRootImpl、WindowManger.LayoutParams 和正在被删除的 View。

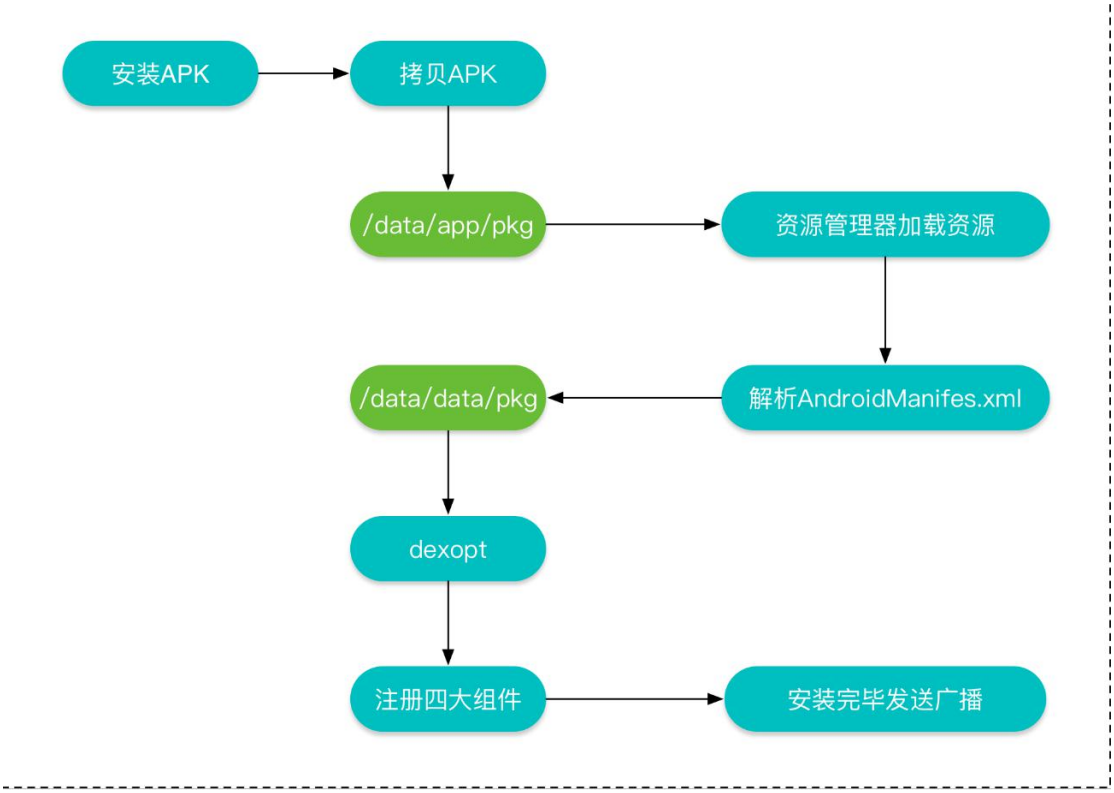
5.Window 的实体是存在于远端的 WindowMangerService，所以增删改 Window 在本端是修改上面的几个 List 然后通过 ViewRootImpl 重绘 View，通过 WindowSession(每 Window 个对应一个)在远端修改 Window。

6.Activity 创建 Window：Activity 会在 attach()中创建 Window 并设置其回调(onAttachedToWindow()、dispatchTouchEvent())，Activity 的 Window 是由 Policy 类创建 PhoneWindow 实现的。然后通过 Activity#setContentView()调用 PhoneWindow 的 setContentView。

13、WMS 是如何管理 Window 的？

14、大体说清一个应用程序安装到手机上时发生了什么？

APK 的安装流程如下所示：



复制 APK 到/data/app 目录下，解压并扫描安装包。

资源管理器解析 APK 里的资源文件。

解析 AndroidManifest 文件，并在/data/data/目录下创建对应的应用数据目录。

然后对 dex 文件进行优化，并保存在 dalvik-cache 目录下。

将 AndroidManifest 文件解析出的四大组件信息注册到 PackageManagerService 中。

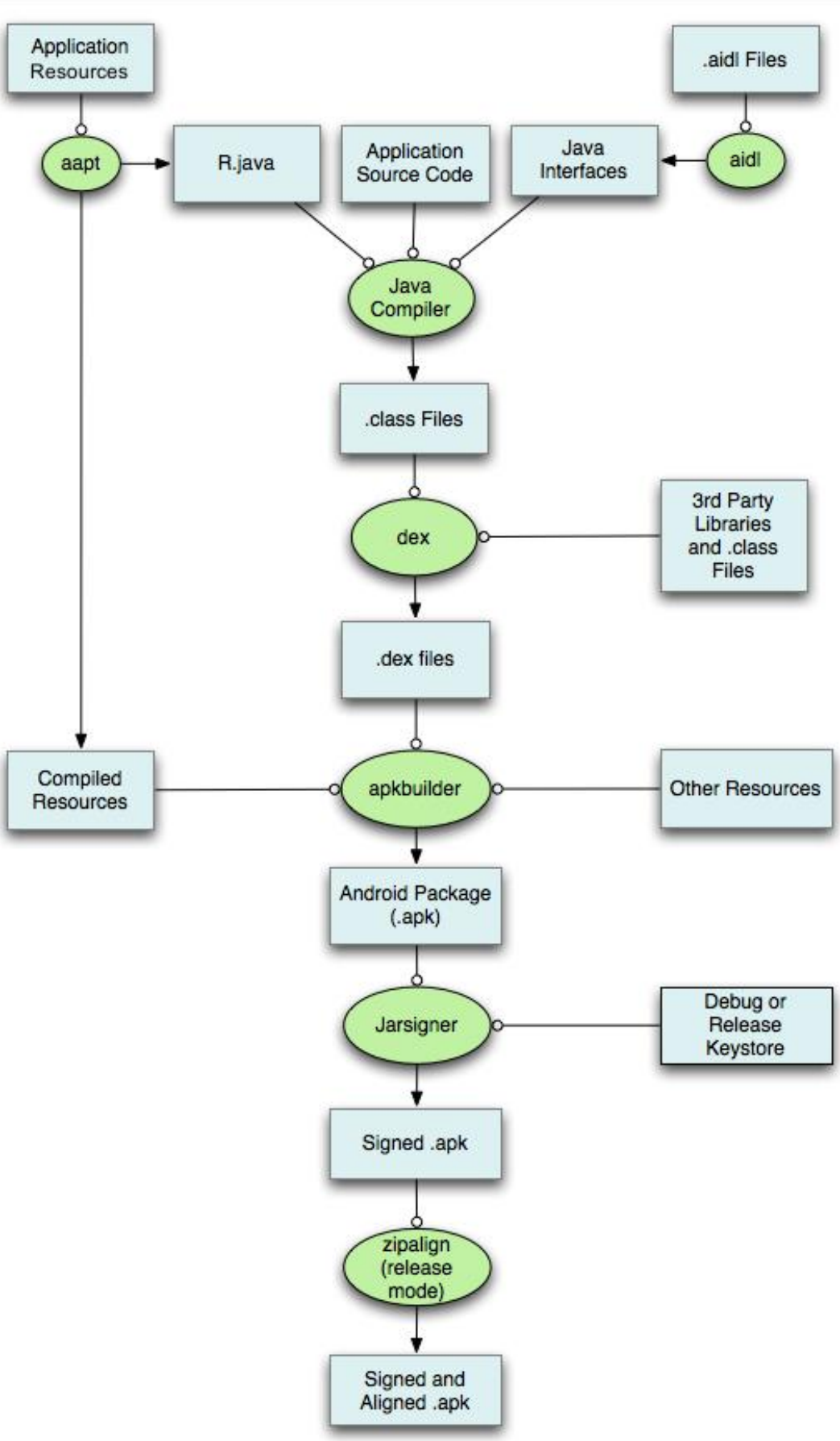
安装完成后，发送广播。

15、Android 的打包流程？（即描述清点击 Android Studio 的 build 按钮后发生了什么？）apk 里有哪些东西？签名算法的原理？

apk 打包流程

Android 的包文件 APK 分为两个部分：代码和资源，所以打包方面也分为资源打包和代码打包两个方面，下面就来分析资源和代码的编译打包原理。

APK 整体的的打包流程如下图所示：


Application Resources
.aidl Files
aapt
R.java
Application Source Code
Java Interfaces
aidl
Java Compiler
.class Files
dex
3rd Party Libraries and .class Files
.dex files
Compiled Resources
apkbuilder
Other Resources
Android Package (.apk)
Jarsigner
Debug or Release Keystore
Signed .apk
zipalign (release mode)
Signed and Aligned .apk

具体说来：

- 通过 AAPT 工具进行资源文件（包括 AndroidManifest.xml、布局文件、各种 xml 资源等）的打包，生成 R.java 文件。
- 通过 AIDL 工具处理 AIDL 文件，生成相应的 Java 文件。
- 通过 Java Compiler 编译 R.java、Java 接口文件、Java 源文件，生成.class 文件。
- 通过 dex 命令，将.class 文件和第三方库中的.class 文件处理生成 classes.dex，该过程主要完成 Java 字节码转换成 Dalvik 字节码，压缩常量池以及清除冗余信息等工作。
- 通过 ApkBuilder 工具将资源文件、DEX 文件打包生成 APK 文件。
- 通过 Jarsigner 工具，利用 KeyStore 对生成的 APK 文件进行签名。
- 如果是正式版的 APK，还会利用 ZipAlign 工具进行对齐处理，对齐的过程就是将 APK 文件中所有的资源文件距离文件的起始距位置都偏移 4 字节的整数倍，这样通过内存映射访问 APK 文件的速度会更快，并且会减少其在设备上运行时的内存占用。

apk 组成

- dex：最终生成的 Dalvik 字节码。
- res：存放资源文件的目录。
- asserts：额外建立的资源文件夹。
- lib：如果存在的话，存放的是 ndk 编出来的 so 库。
- META-INF：存放签名信息

MANIFEST.MF（清单文件）：其中每一个资源文件都有一个 SHA-256-Digest 签名，MANIFEST.MF 文件的 SHA256（SHA1）并 base64 编码的结果即为 CERT.SF 中的 SHA256-Digest-Manifest 值。

CERT.SF（待签名文件）：除了开头处定义的 SHA256（SHA1）-Digest-Manifest 值，后面几项的值是对 MANIFEST.MF 文件中的每项再次 SHA256 并 base64 编码后的值。

CERT.RSA（签名结果文件）：其中包含了公钥、加密算法等信息。首先对前一步生成的 MANIFEST.MF 使用了 SHA256（SHA1）-RSA 算法，用开发者私钥签名，然后在安装时使用公钥解密。最后，将其与未加密的摘要信息（MANIFEST.MF 文件）进行对比，如果相符，则表明内容没有被修改。

- androidManifest：程序的全局清单配置文件。
- resources.arsc：编译后的二进制资源文件。

签名算法的原理

为什么要签名？

- 确保 Apk 来源的真实性。
- 确保 Apk 没有被第三方篡改。

什么是签名？

在 Apk 中写入一个“指纹”。指纹写入以后，Apk 中有任何修改，都会导致这个指纹无效，Android 系统在安装 Apk 进行签名校验时就会不通过，从而保证了安全性。

数字摘要

对一个任意长度的数据，通过一个 Hash 算法计算后，都可以得到一个固定长度的二进制数据，这个数据就称为“摘要”。

补充：

- 散列算法的基础原理：将数据（如一段文字）运算变为另一固定长度值。
- SHA-1：在密码学中，SHA-1（安全散列算法 1）是一种加密散列函数，它接受输入并产生一个 160 位（20 字节）散列值，称为消息摘要 。
- MD5：MD5 消息摘要算法（英语：MD5 Message-Digest Algorithm），一种被广泛使用的密码散列函数，可以产生出一个 128 位（16 字节）的散列值（hash value），用于确保信息传输完整一致。
- SHA-2：名称来自于安全散列算法 2（英语：Secure Hash Algorithm 2）的缩写，一种密码散列函数算法标准，其下又可再分为六个不同的算法标准，包括了：SHA-224、SHA-256、SHA-384、SHA-512、SHA-512/224、SHA-512/256。

特征：

- 唯一性
- 固定长度：比较常用的 Hash 算法有 MD5 和 SHA1，MD5 的长度是 128 拉，SHA1 的长度是 160 位。
- 不可逆性

签名和校验的主要过程

签名就是在摘要的基础上再进行一次加密，对摘要加密后的数据就可以当作数字签名。

签名过程：

- 1、计算摘要：通过 Hash 算法提取出原始数据的摘要。

- 2、计算签名：再通过基于密钥（私钥）的非对称加密算法对提取出的摘要进行加密，加密后的数据就是签名信息。
- 3、写入签名：将签名信息写入原始数据的签名区块内。

校验过程：

- 1、首先用同样的 Hash 算法从接收到的数据中提取出摘要。
- 2、解密签名：使用发送方的公钥对数字签名进行解密，解密出原始摘要。
- 3、比较摘要：如果解密后的数据和提取的摘要一致，则校验通过；如果数据被第三方篡改过，解密后的数据和摘要将会不一致，则校验不通过。

数字证书

如何保证公钥的可靠性呢？答案是数字证书，数字证书是身份认证机构（Certificate Authority）颁发的，包含了以下信息：

- 证书颁发机构
- 证书颁发机构签名
- 证书绑定的服务器域名
- 证书版本、有效期
- 签名使用的加密算法（非对称算法，如 RSA）
- 公钥等

接收方收到消息后，先向 CA 验证证书的合法性，再进行签名校验。

注意：Apk 的证书通常是自签名的，也就是由开发者自己制作，没有向 CA 机构申请。Android 在安装 Apk 时并没有校验证书本身的合法性，只是从证书中提取公钥和加密算法，这也正是对第三方 Apk 重新签名后，还能够继续在没有安装这个 Apk 的系统中继续安装的原因。

keystore 和证书格式

keystore 文件中包含了私钥、公钥和数字证书。根据编码不同，keystore 文件分为很多种，Android 使用的是 Java 标准 keystore 格式 JKS(Java Key Storage)，所以通过 Android Studio 导出的 keystore 文件是以.jks 结尾的。

keystore 使用的证书标准是 X.509，X.509 标准也有多种编码格式，常用的有两种：pem（Privacy Enhanced Mail）和 der（Distinguished Encoding Rules）。jks 使用的是 der 格式，Android 也支持直接使用 pem 格式的证书进行签名。

两种证书编码格式的区别：

- DER（Distinguished Encoding Rules）

二进制格式，所有类型的证书和私钥都可以存储为 der 格式。

- PEM（Privacy Enhanced Mail）

base64 编码，内容以-----BEGIN xxx----- 开头，以-----END xxx----- 结尾。

jarsigner 和 apksigner 的区别

Android 提供了两种对 Apk 的签名方式，一种是基于 JAR 的签名方式，另一种是基于 Apk 的签名方式，它们的主要区别在于使用的签名文件不一样：jarsigner 使用 keystore 文件进行签名；apksigner 除了支持使用 keystore 文件进行签名外，还支持直接指定 pem 证书文件和私钥进行签名。

在签名时，除了要指定 keystore 文件和密码外，也要指定 alias 和 key 的密码，这是为什么呢？

keystore 是一个密钥库，也就是说它可以存储多对密钥和证书，keystore 的密码是用于保护 keystore 本身的，一对密钥和证书是通过 alias 来区分的。所以 jarsigner 是支持使用多个证书对 Apk 进行签名的，apksigner 也同样支持。

Android Apk V1 签名原理

- 1、解析出 CERT.RSA 文件中的证书、公钥，解密 CERT.RSA 中的加密数据。
- 2、解密结果和 CERT.SF 的指纹进行对比，保证 CERT.SF 没有被篡改。
- 3、而 CERT.SF 中的内容再和 MANIFEST.MF 指纹对比，保证 MANIFEST.MF 文件没有被篡改。
- 4、MANIFEST.MF 中的内容和 APK 所有文件指纹逐一对比，保证 APK 没有被篡改。

16、说下安卓虚拟机和 java 虚拟机的原理和不同点?（JVM、Davilk、ART 三者的原理和区别）

JVM 和 Dalvik 虚拟机的区别

JVM:.java -> javac -> .class -> jar -> .jar

架构: 堆和栈的架构.

DVM:.java -> javac -> .class -> dx.bat -> .dex

架构: 寄存器(cpu 上的一块高速缓存)

Android2 个虚拟机的区别（一个 5.0 之前，一个 5.0 之后）

什么是 Dalvik：Dalvik 是 Google 公司自己设计用于 Android 平台的 Java 虚拟机。Dalvik 虚拟机是 Google 等厂商合作开发的 Android 移动设备平台的核心组成部分之一，它可以支持已转换为.dex(即 Dalvik Executable)格式的 Java 应用程序的

运行，.dex 格式是专为 Dalvik 应用设计的一种压缩格式，适合内存和处理器速度有限的系统。Dalvik 经过优化，允许在有限的内存中同时运行多个虚拟机的实例，并且每一个 Dalvik 应用作为独立的 Linux 进程执行。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。

什么是 ART:Android 操作系统已经成熟，Google 的 Android 团队开始将注意力转向一些底层组件，其中之一是负责应用程序运行的 Dalvik 运行时。Google 开发者已经花了两年时间开发更快执行效率更高更省电的替代 ART 运行时。ART 代表 Android Runtime,其处理应用程序执行的方式完全不同于 Dalvik，Dalvik 是依靠一个 Just-In-Time(JIT)编译器去解释字节码。开发者编译后的应用代码需要通过一个解释器在用户的设备上运行，这一机制并不高效，但让应用能更容易在不同硬件和架构上运行。ART 则完全改变了这套做法，在应用安装的时候就预编译字节码为机器语言，这一机制叫 Ahead-Of-Time(AOT)编译。在移除解释代码这一过程后，应用程序执行将更有效率，启动更快。

ART 优点：

- 系统性能的显著提升。
- 应用启动更快、运行更快、体验更流畅、触感反馈更及时。
- 更长的电池续航能力。
- 支持更低的硬件。

ART 缺点：

- 更大的存储空间占用，可能会增加 10%-20%。
- 更长的应用安装时间。

ART 和 Davlik 中垃圾回收的区别？

17、安卓采用自动垃圾回收机制，请说下安卓内存管理的原理？

开放性问题：如何设计垃圾回收算法？

18、Android 中 App 是如何沙箱化的,为何要这么做？

19、一个图片在 app 中调用 R.id 后是如何找到的？

20、JNI

Java 调用 C++

- 在 Java 中声明 Native 方法（即需要调用的本地方法）
- 编译上述 Java 源文件 javac（得到 .class 文件） 3。 通过 javah 命令导出 JNI 的头文件（.h 文件）
- 使用 Java 需要交互的本地代码 实现在 Java 中声明的 Native 方法
- 编译.so 库文件
- 通过 Java 命令执行 Java 程序，最终实现 Java 调用本地代码

C++调用 Java

从 classpath 路径下搜索 ClassMethod 这个类，并返回该类的 Class 对象。

获取类的默认构造方法 ID。

查找实例方法的 ID。

创建该类的实例。

调用对象的实例方法。

```
JNIEXPORT void JNICALL
Java_com_study_jnilearn_AccessMethod_callJavaInstaceMethod

(JNIEnv *env, jclass cls)

{

  jclass clazz = NULL;
```

```
 jobject jobj = NULL;

 jmethodID mid_construct = NULL;

 jmethodID mid_instance = NULL;

 jstring str_arg = NULL;

 // 1、从 classpath 路径下搜索 ClassMethod 这个类，并返回该类的 Class 对象

 clazz = (*env)->FindClass(env, "com/study/jnilearn/ClassMethod");

 if (clazz == NULL) {

     printf("找不到'com.study.jnilearn.ClassMethod'这个类");

     return;

 }



 // 2、获取类的默认构造方法 ID

 mid_construct = (*env)->GetMethodID(env,clazz, "<init>","()V");

 if (mid_construct == NULL) {

     printf("找不到默认的构造方法");

     return;

 }



 // 3、查找实例方法的 ID

 mid_instance = (*env)->GetMethodID(env, clazz, "callInstanceMethod",
"(Ljava/lang/String;I)V");

 if (mid_instance == NULL) {



     return;

 }
```

```
    // 4、创建该类的实例
    jobj = (*env)->NewObject(env,clazz,mid_construct);
    if (jobj == NULL) {
        printf("在 com.study.jnilearn.ClassMethod 类中找不到
callInstanceMethod 方法");
        return;
    }

    // 5、调用对象的实例方法
    str_arg = (*env)->NewStringUTF(env,"我是实例方法");
    (*env)->CallVoidMethod(env,jobj,mid_instance,str_arg,200);

    // 删除局部引用
    (*env)->DeleteLocalRef(env,clazz);
    (*env)->DeleteLocalRef(env,jobj);
    (*env)->DeleteLocalRef(env,str_arg);
}
```

如何在 jni 中注册 native 函数，有几种注册方式？

so 的加载流程是怎样的，生命周期是怎样的？

这个要从 java 层去看源码分析，是从 ClassLoader 的 PathList 中去找到目标路径加载的，同时 so 是通过 mmap 加载映射到虚拟空间的。生命周期加载库和卸载库时分别调用 JNI_OnLoad 和 JNI_OnUnload() 方法。

21、请介绍一下 NDK？

三、Android 优秀三方库源码

1、你项目中用到哪些开源库？说说其实现原理？

一、网络底层框架：OkHttp 实现原理

这个库是做什么用的？

网络底层库，它是基于 http 协议封装的一套请求客户端，虽然它也可以开线程，但根本上它更偏向真正的请求，跟 HttpClient, HttpUrlConnection 的职责是一样的。其中封装了网络请求 get、post 等底层操作的实现。

为什么要在项目中使用这个库？

- OkHttp 提供了对最新的 HTTP 协议版本 HTTP/2 和 SPDY 的支持，这使得对同一个主机发出的所有请求都可以共享相同的套接字连接。
- 如果 HTTP/2 和 SPDY 不可用，OkHttp 会使用连接池来复用连接以提高效率。
- OkHttp 提供了对 GZIP 的默认支持来降低传输内容的大小。
- OkHttp 也提供了对 HTTP 响应的缓存机制，可以避免不必要的网络请求。
- 当网络出现问题时，OkHttp 会自动重试一个主机的多个 IP 地址。

这个库都有哪些用法？对应什么样的使用场景？

get、post 请求、上传文件、上传表单等等。

这个库的优缺点是什么，跟同类型库的比较？

- 优点：在上面
- 缺点：使用的时候仍然需要自己再做一层封装。

这个库的核心实现原理是什么？如果让你实现这个库的某些核心功能，你会考虑怎么去实现？

OkHttp 内部的请求流程：使用 OkHttp 会在请求的时候初始化一个 Call 的实例，然后执行它的 execute()方法或 enqueue()方法，内部最后都会执行到

getResponseWithInterceptorChain()方法，这个方法里面通过拦截器组成的责任链，依次经过用户自定义普通拦截器、重试拦截器、桥接拦截器、缓存拦截器、连接拦截器和用户自定义网络拦截器以及访问服务器拦截器等拦截处理过程，来获取到一个响应并交给用户。其中，除了 OKHttp 的内部请求流程这点之外，缓存和连接这两部分内容也是两个很重要的点，掌握了这 3 点就说明你理解了 OkHttp。

各个拦截器的作用：

- interceptors：用户自定义拦截器
- retryAndFollowUpInterceptor：负责失败重试以及重定向
- BridgeInterceptor：请求时，对必要的 Header 进行一些添加，接收响应时，移除必要的 Header
- CacheInterceptor：负责读取缓存直接返回（根据请求的信息和缓存的响应的信息来判断是否存在缓存可用）、更新缓存
- ConnectInterceptor：负责和服务器建立连接

ConnectionPool：

1、判断连接是否可用，不可用则从 ConnectionPool 获取连接，ConnectionPool 无连接，创建新连接，握手，放入 ConnectionPool。

2、它是一个 Deque，add 添加 Connection，使用线程池负责定时清理缓存。

3、使用连接复用省去了进行 TCP 和 TLS 握手的一个过程。

- networkInterceptors：用户定义网络拦截器

- CallServerInterceptor：负责向服务器发送请求数据、从服务器读取响应数据

你从这个库中学到什么有价值的或者说可借鉴的设计思想？

使用责任链模式实现拦截器的分层设计，每一个拦截器对应一个功能，充分实现了功能解耦，易维护。

手写拦截器？

OKhttp 针对网络层有哪些优化？

网络请求缓存处理，okhttp 如何处理网络缓存的？

HttpUrlConnection 和 okhttp 关系？

Volley 与 OkHttp 的对比：

Volley：支持 HTTPS。缓存、异步请求，不支持同步请求。协议类型是 Http/1.0, Http/1.1，网络传输使用的是 HttpUrlConnection/HttpClient，数据读写使用的 IO。 OkHttp：支持 HTTPS。缓存、异步请求、同步请求。协议类型是 Http/1.0, Http/1.1, SPDY, Http/2.0, WebSocket，网络传输使用的是封装的 Socket，数据读写使用的 NIO（Okio）。 SPDY 协议类似于 HTTP，但旨在缩短网页的加载时间和提高安全性。SPDY 协议通过压缩、多路复用和优先级来缩短加载时间。

Okhttp 的子系统层级结构图如下所示：

网络配置层：利用 Builder 模式配置各种参数，例如：超时时间、拦截器等，这些参数都会由 Okhttp 分发给各个需要的子系统。 重定向层：负责重定向。 Header 拼接层：负责把用户构造的请求转换为发送给服务器的请求，把服务器

返回的响应转换为对用户友好的响应。 HTTP 缓存层：负责读取缓存以及更新缓存。 连接层：连接层是一个比较复杂的层级，它实现了网络协议、内部的拦截器、安全性认证，连接与连接池等功能，但这一层还没有发起真正的连接，它只是做了连接器一些参数的处理。数据响应层：负责从服务器读取响应的数据。在整个 Okhttp 的系统中，我们还要理解以下几个关键角色：

OkHttpClient：通信的客户端，用来统一管理发起请求与解析响应。 Call：Call 是一个接口，它是 HTTP 请求的抽象描述，具体实现类是 RealCall，它由 CallFactory 创建。 Request：请求，封装请求的具体信息，例如：url、header 等。 RequestBody：请求体，用来提交流、表单等请求信息。 Response：HTTP 请求的响应，获取响应信息，例如：响应 header 等。 ResponseBody：HTTP 请求的响应体，被读取一次以后就会关闭，所以我们重复调用 responseBody.string()获取请求结果是会报错的。 Interceptor：Interceptor 是请求拦截器，负责拦截并处理请求，它将网络请求、缓存、透明压缩等功能都统一起来，每个功能都是一个 Interceptor，所有的 Interceptor 最 终连接成一个 Interceptor.Chain。典型的责任链模式实现。 StreamAllocation：用来控制 Connections 与 Streas 的资源分配与释放。 RouteSelector：选择路线与自动重连。 RouteDatabase：记录连接失败的 Route 黑名单。

自己去设计网络请求框架，怎么做？

从网络加载一个 10M 的图片，说下注意事项？

http 怎么知道文件过大是否传输完毕的响应？

谈谈你对 WebSocket 的理解？

WebSocket 与 socket 的区别？

二、网络封装框架：Retrofit 实现原理

这个库是做什么用的？

Retrofit 是一个 RESTful 的 HTTP 网络请求框架的封装。Retrofit 2.0 开始内置 OkHttp，前者专注于接口的封装，后者专注于网络请求的高效。

为什么要在项目中使用这个库？

1、功能强大：

- 支持同步、异步
- 支持多种数据的解析 & 序列化格式
- 支持 RxJava

2、简洁易用：

- 通过注解配置网络请求参数
- 采用大量设计模式简化使用

3、可扩展性好：

- 功能模块高度封装
- 解耦彻底，如自定义 Converters

这个库都有哪些用法？对应什么样的使用场景？

任何网络场景都应该优先选择，特别是后台 API 遵循 Restful API 设计风格 & 项目中使用到 RxJava。

这个库的优缺点是什么，跟同类型库的比较？

- 优点：在上面
- 缺点：扩展性差，高度封装所带来的必然后果，如果服务器不能给出统一的 API 形式，会很难处理。

这个库的核心实现原理是什么？如果让你实现这个库的某些核心功能，你会考虑怎么去实现？

Retrofit 主要是在 create 方法中采用动态代理模式（通过访问代理对象的方式来间接访问目标对象）实现接口方法，这个过程构建了一个 ServiceMethod 对象，根据方法注解获取请求方式，参数类型和参数注解拼接请求的链接，当一切都准备好之后会把数据添加到 Retrofit 的 RequestBuilder 中。然后当我们主动发起网络请求的时候会调用 okhttp 发起网络请求，okhttp 的配置包括请求方式，URL 等在 Retrofit 的 RequestBuilder 的 build()方法中实现，并发起真正的网络请求。

你从这个库中学到什么有价值的或者说可借鉴的设计思想？

内部使用了优秀的架构设计和大量的设计模式，在我分析过 Retrofit 最新版的源码和大量优秀的 Retrofit 源码分析文章后，我发现，要想真正理解 Retrofit 内部的核心源码流程和设计思想，首先，需要对它使用到的九大设计模式有一定的了解，下面我简单说一说：

1、创建 Retrofit 实例：

- 使用建造者模式通过内部 Builder 类建立了一个 Retroift 实例。
- 网络请求工厂使用了工厂方法模式。

2、创建网络请求接口的实例：

- 首先，使用外观模式统一调用创建网络请求接口实例和网络请求参数配置的方法。
- 然后，使用动态代理动态地去创建网络请求接口实例。
- 接着，使用了建造者模式 & 单例模式创建了 serviceMethod 对象。
- 再者，使用了策略模式对 serviceMethod 对象进行网络请求参数配置，即通过解析网络请求接口方法的参数、返回值和注解类型，从 Retrofit 对象

中获取对应的网络的 url 地址、网络请求执行器、网络请求适配器和数据转换器。

- 最后，使用了装饰者模式 ExecuteCallBack 为 serviceMethod 对象加入线程切换的操作，便于接受数据后通过 Handler 从子线程切换到主线程从而对返回数据结果进行处理。

3、发送网络请求：

- 在异步请求时，通过静态 delegate 代理对网络请求接口的方法中的每个参数使用对应的 ParameterHanlder 进行解析。

4、解析数据

5、切换线程：

- 使用了适配器模式通过检测不同的 Platform 使用不同的回调执行器，然后使用回调执行器切换线程，这里同样是使用了装饰模式。

6、处理结果

Android：主流网络请求开源库的对比（Android-Async-Http、Volley、OkHttp、Retrofit）

https://www.jianshu.com/p/050c6db5af5a

三、响应式编程框架：RxJava 实现原理

RxJava 变换操作符 map flatMap concatMap buffer?

- map：【数据类型转换】将被观察者发送的事件转换为另一种类型的事件。

- flatMap：【化解循环嵌套和接口嵌套】将被观察者发送的事件序列进行拆分 & 转换 后合并成一个新的事件序列，最后再进行发送。
- concatMap：【有序】与 flatMap 的 区别在于，拆分 & 重新合并生成的事件序列 的顺序与被观察者旧序列生产的顺序一致。
- buffer：定期从被观察者发送的事件中获取一定数量的事件并放到缓存区中，然后把这些数据集合打包发射。

RxJava 中 map 和 flatmap 操作符的区别及底层实现

手写 rxjava 遍历数组。

你认为 Rxjava 的线程池与你们自己实现任务管理框架有什么区别？

四、图片加载框架：Glide 实现原理

这个库是做什么用的？

Glide 是 Android 中的一个图片加载库，用于实现图片加载。

为什么要在项目中使用这个库？

1、多样化媒体加载：不仅可以进行图片缓存，还支持 Gif、WebP、缩略图，甚至是 Video。

2、通过设置绑定生命周期：可以使加载图片的生命周期动态管理起来。

3、高效的缓存策略：支持内存、Disk 缓存，并且 Picasso 只会缓存原始尺寸的图片，内 Glide 缓存的是多种规格，也就是 Glide 会根据你 ImageView 的大小来缓存相应大小的图片尺寸。

4、内存开销小：默认的 Bitmap 格式是 RGB_565 格式，而 Picasso 默认的是 ARGB_8888 格式，内存开销小一半。

这个库都有哪些用法？对应什么样的使用场景？

1、图片加载：Glide.with(this).load(imageUrl).override(800, 800).placeholder().error().animate().into()。

2、多样式媒体加载：asBitamp、asGif。

3、生命周期集成。

4、可以配置磁盘缓存策略 ALL、NONE、SOURCE、RESULT。

这个库的优缺点是什么，跟同类型库的比较？

库比较大，源码实现复杂。

这个库的核心实现原理是什么？如果让你实现这个库的某些核心功能，你会考虑怎么去实现？

- Glide&with：

1、初始化各式各样的配置信息（包括缓存，请求线程池，大小，图片格式等等）以及 glide 对象。

2、将 glide 请求和 application/SupportFragment/Fragment 的生命周期绑定在一块。

- Glide&load：

设置请求 url，并记录 url 已设置的状态。

3、Glide&into：

1、首先根据转码类 transcodeClass 类型返回不同的 ImageViewTarget：BitmapImageViewTarget、DrawableImageViewTarget。

2、递归建立缩略图请求，没有缩略图请求，则直接进行正常请求。

3、如果没指定宽高，会根据 ImageView 的宽高计算出图片宽高，最终执行到 onSizeReay()方法中的 engine.load()方法。

4、engine 是一个负责加载和管理缓存资源的类

- 常规三级缓存的流程：强引用->软引用->硬盘缓存

当我们的 APP 中想要加载某张图片时，先去 LruCache 中寻找图片，如果 LruCache 中有，则直接取出来使用，如果 LruCache 中没有，则去 SoftReference 中寻找（软引用适合当 cache，当内存吃紧的时候才会被回收。而 weakReference 在每次 system.gc（）就会被回收）（当 LruCache 存储紧张时，会把最近最少使用的数据放到 SoftReference 中），如果 SoftReference 中有，则从 SoftReference 中取出图片使用，同时将图片重新放回到 LruCache 中，如果 SoftReference 中也没有图片，则去硬盘缓存中中寻找，如果有则取出来使用，同时将图片添加到 LruCache 中，如果没有，则连接网络从网上下载图片。图片下载完成后，将图片保存到硬盘缓存中，然后放到 LruCache 中。

- Glide 的三层缓存机制：

Glide 缓存机制大致分为三层：内存缓存、弱引用缓存、磁盘缓存。

取的顺序是：内存、弱引用、磁盘。
存的顺序是：弱引用、内存、磁盘。

三层存储的机制在 Engine 中实现的。先说下 Engine 是什么？Engine 这一层负责加载时做管理内存缓存的逻辑。持有 MemoryCache、Map<Key,

WeakReference<EngineResource<?>>>。通过 load（）来加载图片，加载前后会做内存存储的逻辑。如果内存缓存中没有，那么才会使用 EngineJob 这一层来进行异步获取硬盘资源或网络资源。EngineJob 类似一个异步线程或 observable。Engine 是一个全局唯一的，通过 Glide.getEngine()来获取。

需要一个图片资源，如果 Lrucache 中有相应的资源图片，那么就返回，同时从 Lrucache 中清除，放到 activeResources 中。activeResources map 是盛放正在使用的资源，以弱引用的形式存在。同时资源内部有被引用的记录。如果资源没有引用记录了，那么再放回 Lrucache 中，同时从 activeResources 中清除。如果 Lrucache 中没有，就从 activeResources 中找，找到后相应资源引用加 1。如果 Lrucache 和 activeResources 中没有，那么进行资源异步请求（网络/diskLrucache），请求成功后，资源放到 diskLrucache 和 activeResources 中。

Glide 源码机制的核心思想：

使用一个弱引用 map activeResources 来盛放项目中正在使用的资源。Lrucache 中不含有正在使用的资源。资源内部有个计数器来显示自己是不是还有被引用的情况，把正在使用的资源和没有被使用的资源分开有什么好处呢？？因为当 Lrucache 需要移除一个缓存时，会调用 resource.recycle()方法。注意到该方法上面注释写着只有没有任何 consumer 引用该资源的时候才可以调用这个方法。那么为什么调用 resource.recycle()方法需要保证该资源没有任何 consumer 引用呢？glide 中 resource 定义的 recycle（）要做的事情是把这个不用的资源（假设是 bitmap 或 drawable）放到 bitmapPool 中。bitmapPool 是一个 bitmap 回收再利用的库，在做 transform 的时候会从这个 bitmapPool 中拿一个 bitmap 进行再利用。这样就避免了重新创建 bitmap，减少了内存的开支。而既然 bitmapPool 中的 bitmap 会被重复利用，那么肯定要保证回收该资源的时候（即调用资源的

recycle（）时），要保证该资源真的没有外界引用了。这也是为什么 glide 花费那么多逻辑来保证 Lrucache 中的资源没有外界引用的原因。

你从这个库中学到什么有价值的或者说可借鉴的设计思想？

Glide 的高效的三层缓存机制，如上。

Glide 如何确定图片加载完毕？

Glide 使用什么缓存？

Glide 内存缓存如何控制大小？

计算一张图片的大小

图片占用内存的计算公式：图片高度 * 图片宽度 * 一个像素占用的内存大小。所以，计算图片占用内存大小的时候，要考虑图片所在的目录跟设备密度，这两个因素其实影响的是图片的宽高，android 会对图片进行拉升跟压缩。

加载 bitmap 过程（怎样保证不产生内存溢出）

由于 Android 对图片使用内存有限制，若是加载几兆的大图片便内存溢出。Bitmap 会将图片的所有像素（即长 x 宽）加载到内存中，如果图片分辨率过大，会直接导致内存 OOM，只有在 BitmapFactory 加载图片时使用 BitmapFactory.Options 对相关参数进行配置来减少加载的像素。

BitmapFactory.Options 相关参数详解：

(1).Options.inPreferredConfig 值来降低内存消耗。

比如：默认值 ARGB_8888 改为 RGB_565,节约一半内存。

(2).设置 Options.inSampleSize 缩放比例，对大图片进行压缩 。

(3).设置 Options.inPurgeable 和 inInputShareable：让系统能及时回收内存。

A：inPurgeable：设置为 True 时，表示系统内存不足时可以被回收，设置为 False 时，表示不能被回收。

B：inInputShareable：设置是否深拷贝，与 inPurgeable 结合使用，inPurgeable 为 false 时，该参数无意义。

(4).使用 decodeStream 代替 decodeResource 等其他方法。

Android 中软引用与弱引用的应用场景。

Java 引用类型分类：

在 Android 应用的开发中，为了防止内存溢出，在处理一些占用内存大而且生命周期较长的对象时候，可以尽量应用软引用和弱引用技术。

- 1、软/弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。利用这个队列可以得知被回收的软/弱引用的对象列表，从而为缓冲器清除已失效的软 / 弱引用。
- 2、如果只是想避免 OOM 异常的发生，则可以使用软引用。如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。
- 3、可以根据对象是否经常使用来判断选择软引用还是弱引用。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

Android 里的内存缓存和磁盘缓存是怎么实现的。

内存缓存基于 LruCache 实现，磁盘缓存基于 DiskLruCache 实现。这两个类都基于 Lru 算法和 LinkedHashMap 来实现。

LRU 算法可以用一句话来描述，如下所示：

LRU 是 Least Recently Used 的缩写，最近最少使用算法，从它的名字就可以看出，它的核心原则是如果一个数据在最近一段时间没有使用到，那么它在将来被访问到的可能性也很小，则这类数据项会被优先淘汰掉。

LruCache 原理

之前，我们会使用内存缓存技术实现，也就是软引用或弱引用，在 Android 2.3（APILevel 9）开始，垃圾回收器会更倾向于回收持有软引用或弱引用的对象，这让软引用和弱引用变得不再可靠。

其实 LRU 缓存的实现类似于一个特殊的栈，把访问过的元素放置到栈顶（若栈中存在，则更新至栈顶；若栈中不存在则直接入栈），然后如果栈中元素数量超过限定值，则删除栈底元素（即最近最少使用的元素）。

它的内部存在一个 LinkedHashMap 和 maxSize，把最近使用的对象用强引用存储在 LinkedHashMap 中，给出来 put 和 get 方法，每次 put 图片时计算缓存中所有图片的总大小，跟 maxSize 进行比较，大于 maxSize，就将最久添加的图片移除，反之小于 maxSize 就添加进来。

LruCache 的原理就是利用 LinkedHashMap 持有对象的强引用，按照 Lru 算法进行对象淘汰。具体说来假设我们从表尾访问数据，在表头删除数据，当访问的数据项在链表中存在时，则将该数据项移动到表尾，否则在表尾新建一个数据项。当链表容量超过一定阈值，则移除表头的数据。

详细来说就是 LruCache 中维护了一个集合 LinkedHashMap，该 LinkedHashMap 是以访问顺序排序的。当调用 put()方法时，就会在结合中添加元素，并调用 trimToSize()判断缓存是否已满，如果满了就用 LinkedHashMap 的迭代器删除队头元素，即近期最少访问的元素。当调用 get()方法访问缓存对象时，就会调用 LinkedHashMap 的 get()方法获得对应集合元素，同时会更新该元素到队尾。

LruCache put 方法核心逻辑

在添加过缓存对象后，调用 trimToSize()方法，来判断缓存是否已满，如果满了就要删除近期最少使用的对象。trimToSize()方法不断地删除 LinkedHashMap 中队头的元素，即近期最少访问的，直到缓存大小小于最大值（maxSize）。

LruCache get 方法核心逻辑

当调用 LruCache 的 get()方法获取集合中的缓存对象时，就代表访问了一次该元素，将会更新队列，保持整个队列是按照访问顺序排序的。

为什么会选择 LinkedHashMap 呢？

这跟 LinkedHashMap 的特性有关，LinkedHashMap 的构造函数里有个布尔参数 accessOrder，当它为 true 时，LinkedHashMap 会以访问顺序为序排列元素，否则以插入顺序为序排序元素。

LinkedHashMap 原理

LinkedHashMap 几乎和 HashMap 一样：从技术上来说，不同的是它定义了一个 Entry<K,V> header，这个 header 不是放在 Table 里，它是额外独立出来的。LinkedHashMap 通过继承 hashMap 中的 Entry<K,V>,并添加两个属性

Entry<K,V> before,after,和 header 结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。

DisLruCache 原理

DiskLruCache 与 LruCache 原理相似，只是多了一个 journal 文件来做磁盘文件的管理，如下所示：

```
libcore.io.DiskLruCache

1

1

1



DIRTY 1517126350519

CLEAN 1517126350519 5325928

REMOVE 1517126350519
```

注：这里的缓存目录是应用的缓存目录/data/data/pckagename/cache，未 root 的手机可以通过以下命令进入到该目录中或者将该目录整体拷贝出来：

```
//进入/data/data/pckagename/cache 目录

adb shell

run-as com.your.packagename

cp /data/data/com.your.packagename/



//将/data/data/pckagename 目录拷贝出来

adb backup -noapk com.your.packagename
```

我们来分析下这个文件的内容：

第一行：libcore.io.DiskLruCache，固定字符串。 第二行：1，DiskLruCache 源码版本号。 第三行：1，App 的版本号，通过 open()方法传入进去的。 第四行：1，每个 key 对应几个文件，一般为 1. 第五行：空行 第六行及后续行：缓存操作记录。 第六行及后续行表示缓存操作记录，关于操作记录，我们需要了解以下三点：

DIRTY 表示一个 entry 正在被写入。写入分两种情况，如果成功会紧接着写入一行 CLEAN 的记录；如果失败，会增加一行 REMOVE 记录。注意单独只有 DIRTY 状态的记录是非法的。 当手动调用 remove(key)方法的时候也会写入一条 REMOVE 记录。 READ 就是说明有一次读取的记录。 CLEAN 的后面还记录了文件的长度，注意可能会一个 key 对应多个文件，那么就会有多个数字。

Bitmap 压缩策略

加载 Bitmap 的方式：

BitmapFactory 四类方法：

- decodeFile(文件系统)
- decodeResourece(资源)
- decodeStream(输入流)
- decodeByteArray(字节数)

BitmapFactory.options 参数:

- inSampleSize 采样率，对图片高和宽进行缩放，以最小比进行缩放（一般取值为 2 的指数）。通常是根据图片宽高实际的大小/需要的宽高大小，

分别计算出宽和高的缩放比。但应该取其中最小的缩放比，避免缩放图片太小，到达指定控件中不能铺满，需要拉伸从而导致模糊。

- inJustDecodeBounds 获取图片的宽高信息，交给 inSampleSize 参数选择缩放比。通过 inJustDecodeBounds = true，然后加载图片就可以实现只解析图片的宽高信息，并不会真正的加载图片，所以这个操作是轻量级的。当获取了宽高信息，计算出缩放比后，然后在将 inJustDecodeBounds = false,再重新加载图片，就可以加载缩放后的图片。

高效加载 Bitmap 的流程:

- 1、将 BitmapFactory.Options 的 inJustDecodeBounds 参数设为 true 并加载图片
- 2、从 BitmapFactory.Options 中取出图片原始的宽高信息，对应于 outWidth 和 outHeight 参数
- 3、根据采样率规则并结合目标 view 的大小计算出采样率 inSampleSize
- 4、将 BitmapFactory.Options 的 inJustDecodeBounds 设置为 false 重新加载图片

Bitmap 的处理：

当使用 ImageView 的时候，可能图片的像素大于 ImageView，此时就可以通过 BitmapFactory.Option 来对图片进行压缩，inSampleSize 表示缩小 2^(inSampleSize-1)倍。

BitMap 的缓存：

1.使用 LruCache 进行内存缓存。

2.使用 DiskLruCache 进行硬盘缓存。

实现一个 ImageLoader 的流程

同步异步加载、图片压缩、内存硬盘缓存、网络拉取

- 1.同步加载只创建一个线程然后按照顺序进行图片加载
- 2.异步加载使用线程池，让存在的加载任务都处于不同线程
- 3.为了不开启过多的异步任务，只在列表静止的时候开启图片加载

具体为：

- 1、ImageLoader 作为一个单例，提供了加载图片到指定控件的方法：直接从内存缓存中获取对象，如果没有则用一个 ThreadPoolExecutor 去执行 Runnable 任务来加载图片。ThreadPoolExecutor 的创建需要指定核心线程数 CPU 数+1，最大线程数 CPU 数*2+1，线程闲置超市时长 10s,这几个关键数据，还可以加入 ThreadFactory 参数来创建定制化的线程。
- 2、ImageLoader 的具体实现 loadBitmap：先从内存缓存 LruCache 中加载，如果为空再从磁盘缓存中加载，加载成功后记得存入内存缓存，如果为空则从网络中直接下载输出流到磁盘缓存，然后再从磁盘中加载，如果为空并且磁盘缓存没有被创建的话，直接通过 BitmapFactory 的 decodeStream 获取网络请求的输入流获取 Bitmap 对象。
- 3、v4 包的 LruCache 可以兼容到 2.2 版本，LruCache 采用 LinkedHashMap 存储缓存对象。创建对象只需要提供缓存容量并重写 sizeOf 方法：作用是计算缓存对象的大小。有时需要重写 entryRemoved 方法，用于回收一些资源。
- 4、DiskLruCache 通过 open 方法创建，设置缓存路径，缓存容量。缓存添加通过 Editor 对象创建输出流，下载资源到输出流完成后，commit，如果失败则 abort 撤回。然后刷新磁盘缓存。缓存查找通过 Snapshot 对

象获取输入流，获取 FileDescriptor，通过 FileDescriptor 解析出 Bitmap 对象。

- 5、列表中需要加载图片的时候，当列表在滑动中不进行图片加载，当滑动停止后再去加载图片。

Bitmap 在 decode 的时候申请的内存如何复用，释放时机

图片库对比

http://stackoverflow.com/questions/29363321/picasso-v-s-imageloader-v-s-fresco-vs-glide

http://www.trinea.cn/android/android-image-cache-compare/

Fresco 与 Glide 的对比：

Glide：相对轻量级，用法简单优雅，支持 Gif 动态图，适合用在那些对图片依赖不大的 App 中。 Fresco：采用匿名共享内存来保存图片，也就是 Native 堆，有效的的避免了 OOM，功能强大，但是库体积过大，适合用在对图片依赖比较大的 App 中。

Fresco 的整体架构如下图所示：

DraweeView：继承于 ImageView，只是简单的读取 xml 文件的一些属性值和做一些初始化的工作，图层管理交由 Hierarchy 负责，图层数据获取交由负责。 DraweeHierarchy：由多层 Drawable 组成，每层 Drawable 提供某种功能（例如：缩放、圆角）。 DraweeController：控制数据的获取与图片加载，向 pipeline

发出请求，并接收相应事件，并根据不同事件控制 Hierarchy，从 DraweeView 接收用户的事件，然后执行取消网络请求、回收资源等操作。 DraweeHolder：统筹管理 Hierarchy 与 DraweeHolder。 ImagePipeline：Fresco 的核心模块，用来以各种方式（内存、磁盘、网络等）获取图像。 Producer/Consumer：Producer 也有很多种，它用来完成网络数据获取，缓存数据获取、图片解码等多种工作，它产生的结果由 Consumer 进行消费。 IO/Data：这一层便是数据层了，负责实现内存缓存、磁盘缓存、网络缓存和其他 IO 相关的功能。 纵观整个 Fresco 的架构，DraweeView 是门面，和用户进行交互，DraweeHierarchy 是视图层级，管理图层，DraweeController 是控制器，管理数据。它们构成了整个 Fresco 框架的三驾马车。当然还有我们 幕后英雄 Producer，所有的脏活累活都是它干的，最佳劳模

理解了 Fresco 整体的架构，我们还有了解在这套矿建里发挥重要作用的几个关键角色，如下所示：

Supplier：提供一种特定类型的对象，Fresco 里有很多以 Supplier 结尾的类都实现了这个接口。 SimpleDraweeView：这个我们就很熟悉了，它接收一个 URL，然后调用 Controller 去加载图片。该类继承于 GenericDraweeView，GenericDraweeView 又继承于 DraweeView，DraweeView 是 Fresco 的顶层 View 类。 PipelineDraweeController：负责图片数据的获取与加载，它继承于 AbstractDraweeController，由 PipelineDraweeControllerBuilder 构建而来。AbstractDraweeController 实现了 DraweeController 接口，DraweeController 是 Fresco 的数据大管家，所以的图片数据的处理都是由它来完成的。GenericDraweeHierarchy：负责 SimpleDraweeView 上的图层管理，由多层

Drawable 组成，每层 Drawable 提供某种功能（例如：缩放、圆角），该类由 GenericDraweeHierarchyBuilder 进行构建，该构建器 将 placeholderImage、retryImage、failureImage、progressBarImage、background、overlays 与 pressedStateOverlay 等 xml 文件或者 Java 代码里设置的属性信息都传入 GenericDraweeHierarchy 中，由 GenericDraweeHierarchy 进行处理。

DraweeHolder：该类是一个 Holder 类，和 SimpleDraweeView 关联在一起，DraweeView 是通过 DraweeHolder 来统一管理的。而 DraweeHolder 又是用来统一管理相关的 Hierarchy 与 Controller DataSource：类似于 Java 里的 Futures，代表数据的来源，和 Futures 不同，它可以有多个 result。 DataSubscriber：接收 DataSource 返回的结果。 ImagePipeline：用来调取获取图片的接口。

Producer：加载与处理图片，它有多种实现，例如：NetworkFetcherProducer，LocalAssetFetcherProducer，LocalFileFetchProducer。从这些类的名字我们就可以知道它们是干什么的。 Producer 由 ProducerFactory 这个工厂类构建的，而且所有的 Producer 都是像 Java 的 IO 流那样，可以一层嵌套一层，最终只得到一个结果，这是一个很精巧的设计 Consumer：用来接收 Producer 产生的结果，它与 Producer 组成了生产者与消费者模式。 注：Fresco 源码里的类的名字都比较长，但是都是按照一定的命令规律来的，例如：以 Supplier 结尾的类都实现了 Supplier 接口，它可以提供某一个类型的对象（factory, generator, builder, closure 等）。 以 Builder 结尾的当然就是以构造者模式创建对象的类。

Bitmap 如何处理大图，如一张 30M 的大图，如何预防 OOM?

http://blog.csdn.net/guolin_blog/article/details/9316683

https://blog.csdn.net/lmj623565791/article/details/493009890

使用 BitmapRegionDecoder 动态加载图片的显示区域。

Bitmap 对象的理解。

对 inBitmap 的理解。

自己去实现图片库，怎么做？（对扩展开发，对修改封闭，同时又保持独立性，参考 Android 源码设计模式解析实战的图片加载库案例即可）

写个图片浏览器，说出你的思路？

五、事件总线框架：EventBus 实现原理

六、内存泄漏检测框架：LeakCanary 实现原理

这个库是做什么用？

内存泄露检测框架。

为什么要在项目中使用这个库？

- 针对 Android Activity 组件完全自动化的内存泄漏检查，在最新的版本中，还加入了 android.app.fragment 的组件自动化的内存泄漏检测。
- 易用集成，使用成本低。
- 友好的界面展示和通知。

这个库都有哪些用法？对应什么样的使用场景？

直接从 application 中拿到全局的 refWatcher 对象，在 Fragment 或其他组件的销毁回调中使用 refWatcher.watch(this)检测是否发生内存泄漏。

这个库的优缺点是什么，跟同类型库的比较？

检测结果并不是特别的准确，因为内存的释放和对象的生命周期有关也和 GC 的调度有关。

这个库的核心实现原理是什么？如果让你实现这个库的某些核心功能，你会考虑怎么去实现？

主要分为如下 7 个步骤：

- 1、RefWatcher.watch()创建了一个 KeyedWeakReference 用于去观察对象。
- 2、然后，在后台线程中，它会检测引用是否被清除了，并且是否没有触发 GC。
- 3、如果引用仍然没有被清除，那么它将会把堆栈信息保存在文件系统中的.hprof 文件里。
- 4、HeapAnalyzerService 被开启在一个独立的进程中，并且 HeapAnalyzer 使用了 HAHA 开源库解析了指定时刻的堆栈快照文件 heap dump。
- 5、从 heap dump 中，HeapAnalyzer 根据一个独特的引用 key 找到了 KeyedWeakReference，并且定位了泄露的引用。
- 6、HeapAnalyzer 为了确定是否有泄露，计算了到 GC Roots 的最短强引用路径，然后建立了导致泄露的链式引用。
- 7、这个结果被传回到 app 进程中的 DisplayLeakService，然后一个泄露通知便展现出来了。

简单来说就是：

在一个 Activity 执行完 onDestroy()之后，将它放入 WeakReference 中，然后将这个 WeakReference 类型的 Activity 对象与 ReferenceQueque 关联。这时再从 ReferenceQueque 中查看是否有该对象，如果没有，执行 gc，再次查看，还是没有的话则判断发生内存泄露了。最后用 HAHA 这个开源库去分析 dump 之后

的 heap 内存（主要就是创建一个 HprofParser 解析器去解析出对应的引用内存快照文件 snapshot）。

流程图：

源码分析中一些核心分析点：

AndroidExcludedRefs：它是一个 enum 类，它声明了 Android SDK 和厂商定制的 SDK 中存在的内存泄露的 case，根据 AndroidExcludedRefs 这个类的类名就可看出这些 case 都会被 Leakcanary 的监测过滤掉。

buildAndInstall()（即 install 方法）这个方法应该仅仅只调用一次。

debuggerControl：判断是否处于调试模式，调试模式中不会进行内存泄漏检测。为什么呢？因为在调试过程中可能会保留上一个引用从而导致错误信息上报。

watchExecutor：线程控制器，在 onDestroy() 之后并且主线程空闲时执行内存泄漏检测。

gcTrigger：用于 GC，watchExecutor 首次检测到可能的内存泄漏，会主动进行 GC，GC 之后会再检测一次，仍然泄漏的判定为内存泄漏，最后根据 heapDump 信息生成相应的泄漏引用链。

gcTrigger 的 runGc()方法：这里并没有使用 System.gc()方法进行回收，因为 system.gc()并不会每次都执行。而是从 AOSP 中拷贝一段 GC 回收的代码，从而相比 System.gc()更能够保证进行垃圾回收的工作。

```
Runtime.getRuntime().gc();
```

子线程延时 1000ms；

System.runFinalization();

install 方法内部最终还是调用了 application 的 registerActivityLifecycleCallbacks()方法，这样就能够监听 activity 对应的生命周期事件了。

在 RefWatcher#watch()中使用随机的 UUID 保证了每个检测对象对应的 key 的唯一性。

在 KeyedWeakReference 内部，使用了 key 和 name 标识了一个被检测的 WeakReference 对象。在其构造方法中将弱引用和引用队列 ReferenceQueue 关联起来，如果弱引用 reference 持有的对象被 GC 回收，JVM 就会把这个弱引用加入到与之关联的引用队列 referenceQueue 中。即 KeyedWeakReference 持有的 Activity 对象如果被 GC 回收，该对象就会加入到引用队列 referenceQueue 中。

使用 Android SDK 的 API Debug.dumpHprofData() 来生成 hprof 文件。

在 HeapAnalyzerService（类型为 IntentService 的 ForegroundService）的 runAnalysis()方法中，为了避免减慢 app 进程或占用内存，这里将 HeapAnalyzerService 设置在了一个独立的进程中。

你从这个库中学到什么有价值的或者说可借鉴的设计思想？

leakCannary 中如何判断一个对象是否被回收？如何触发手动 gc？c 层实现？

BlockCanary 原理：

该组件利用了主线程的消息队列处理机制，应用发生卡顿，一定是在 dispatchMessage 中执行了耗时操作。我们通过给主线程的 Looper 设置一个

Printer，打点统计 dispatchMessage 方法执行的时间，如果超出阀值，表示发生卡顿，则 dump 出各种信息，提供开发者分析性能瓶颈。

七、依赖注入框架：ButterKnife 实现原理

ButterKnife 对性能的影响很小，因为没有使用使用反射，而是使用的 Annotation Processing Tool(APT)，注解处理器，javac 中用于编译时扫描和解析 Java 注解的工具。在编译阶段执行的，它的原理就是读入 Java 源代码，解析注解，然后生成新的 Java 代码。新生成的 Java 代码最后被编译成 Java 字节码，注解解析器不能改变读入的 Java 类，比如不能加入或删除 Java 方法。

AOP IOC 的好处以及在 Android 开发中的应用

八、依赖全局管理框架：Dagger2 实现原理

九、数据库框架：GreenDao 实现原理

数据库框架对比？

数据库的优化

数据库数据迁移问题

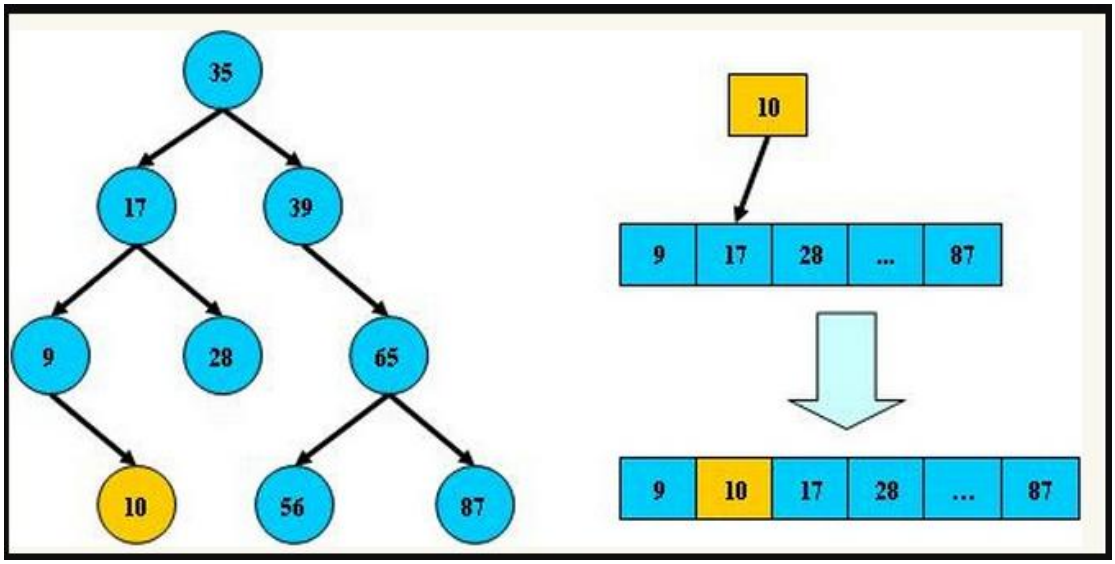
数据库索引的数据结构

平衡二叉树

- 1、非叶子节点只能允许最多两个子节点存在。

- 2、每一个非叶子节点数据分布规则为左边的子节点小当前节点的值，右边的子节点大于当前节点的值(这里值是基于自己的算法规则而定的，比如 hash 值)。
- 3、树的左右两边的层级数相差不会大于 1。

使用平衡二叉树能保证数据的左右两边的节点层级相差不会大于 1.，通过这样避免树形结构由于删除增加变成线性链表影响查询效率，保证数据平衡的情况下查找数据的速度近于二分法查找。



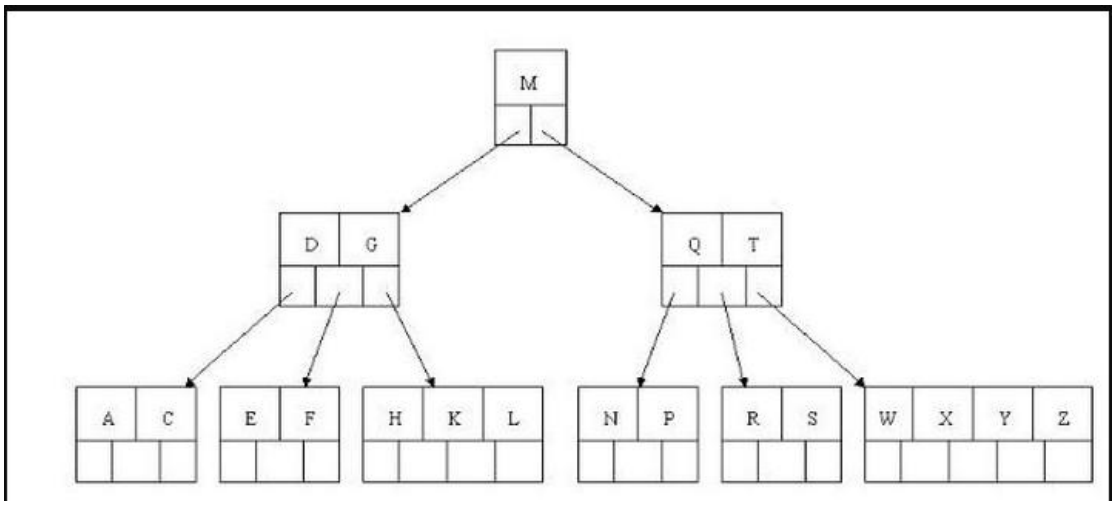
目前大部分数据库系统及文件系统都采用 B-Tree 或其变种 B+Tree 作为索引结构。

B-Tree

B 树和平衡二叉树稍有不同的是 B 树属于多叉树又名平衡多路查找树（查找路径不只两个）。

- 1、排序方式：所有节点关键字是按递增次序排列，并遵循左小右大原则。

- 2、子节点数：非叶节点的子节点数>1，且<=M ，且 M>=2，空树除外（注：M 阶代表一个树节点最多有多少个查找路径，M=M 路,当 M=2 则是 2 叉树,M=3 则是 3 叉）。
- 3、关键字数：枝节点的关键字数量大于等于 ceil(m/2)-1 个且小于等于 M-1 个（注：ceil()是个朝正无穷方向取整的函数 如 ceil(1.1)结果为 2)。
- 4、所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也有指向其子节点的指针只不过其指针地址都为 null 对应下图最后一层节点的空格子。



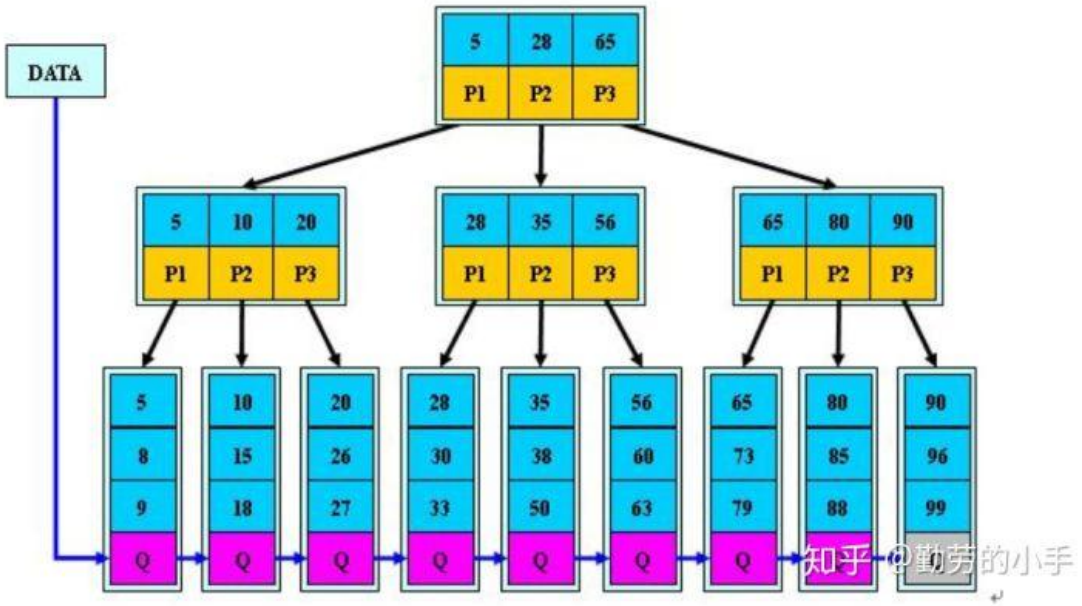
B 树相对于平衡二叉树的不同是，每个节点包含的关键字增多了，把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度。

B+Tree

规则：

- 1、B+跟 B 树不同 B+树的非叶子节点不保存关键字记录的指针，只进行数据索引。
- 2、B+树叶子节点保存了父节点的所有关键字记录的指针，所有数据地址必须要到叶子节点才能获取到。所以每次数据查询的次数都一样。

- 3、B+树叶子节点的关键字从小到大有序排列，左边结尾数据都会保存右边节点开始数据的指针。
- 4、非叶子节点的子节点数=关键字数（来源百度百科）（根据各种资料 这里有两种算法的实现方式，另一种为非叶节点的关键字数=子节点数-1（来源维基百科)，虽然他们数据排列结构不一样，但其原理还是一样的 Mysql 的 B+树是用第一种方式实现）。



特点：

1、B+树的层级更少：相较于 B 树 B+每个非叶子节点存储的关键字数更多，树的层级更少所以查询数据更快。

2、B+树查询速度更稳定：B+所有关键字数据地址都存在叶子节点上，所以每次查找的次数都相同所以查询速度要比 B 树更稳定。

3、B+树天然具备排序功能：B+树所有的叶子节点数据构成了一个有序链表，在查询大小区间的数据时候更方便，数据紧密性很高，缓存的命中率也会比 B 树高。

4、B+树全节点遍历更快：B+树遍历整棵树只需要遍历所有的叶子节点即可，而不需要像 B 树一样需要对每一层进行遍历，这有利于数据库做全表扫描。

B 树相对于 B+树的优点是，如果经常访问的数据离根节点很近，而 B 树的非叶子节点本身存有关键字其数据的地址，所以这种数据检索的时候会要比 B+树快。

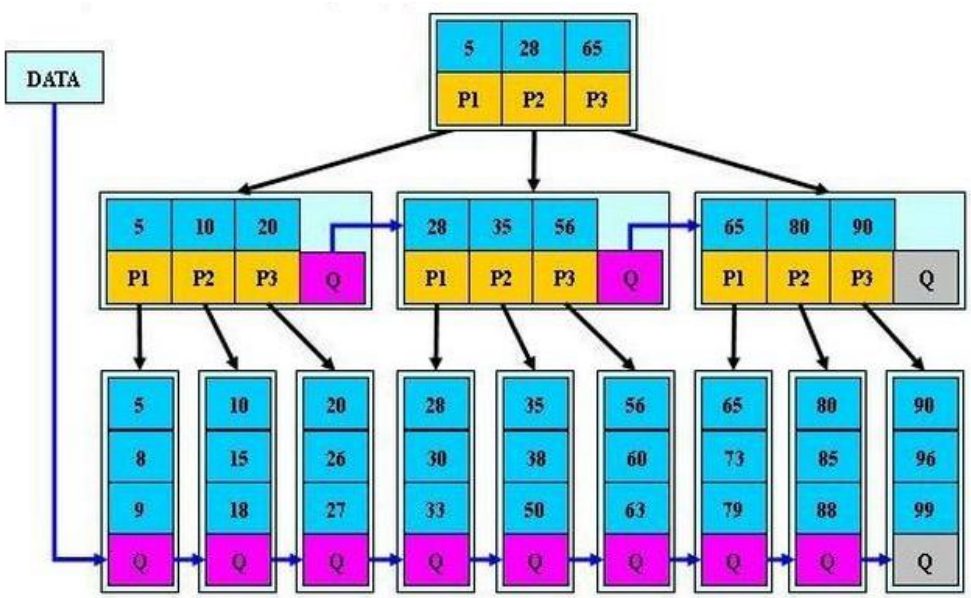
B*Tree

B*树是 B+树的变种，相对于 B+树他们的不同之处如下：

1、首先是关键字个数限制问题，B+树初始化的关键字初始化个数是 cei(m/2)，b 树的初始化个数为（cei(2/3m)）。

2、B+树节点满时就会分裂，而 B*树节点满时会检查兄弟节点是否满（因为每个节点都有指向兄弟的指针），如果兄弟节点未满则向兄弟节点转移关键字，如果兄弟节点已满，则从当前节点和兄弟节点各拿出 1/3 的数据创建一个新的节点出来。

在 B+树的基础上因其初始化的容量变大，使得节点空间使用率更高，而又存有兄弟节点的指针，可以向兄弟节点转移关键字的特性使得 B*树分解次数变得更少。



结论：

- 1、相同思想和策略：从平衡二叉树、B 树、B+树、B*树总体来看它们贯彻的思想是相同的，都是采用二分法和数据平衡策略来提升查找数据的速度。
- 2、不同的方式的磁盘空间利用：不同点是他们一个一个在演变的过程中通过 IO 从磁盘读取数据的原理进行一步步的演变，每一次演变都是为了让节点的空间更合理的运用起来，从而使树的层级减少达到快速查找数据的目的；

还不理解请查看：平衡二叉树、B 树、B+树、B*树 理解其中一种你就都明白了。

四、热修复、插件化、模块化、组件化、Gradle、编译插桩技术

1、热修复和插件化

Android 中 ClassLoader 的种类&特点

- BootClassLoader（Java 的 BootStrap ClassLoader）： 用于加载 Android Framework 层 class 文件。

- PathClassLoader（Java 的 App ClassLoader）： 用于加载已经安装到系统中的 apk 中的 class 文件。
- DexClassLoader（Java 的 Custom ClassLoader）： 用于加载指定目录中的 class 文件。
- BaseDexClassLoader： 是 PathClassLoader 和 DexClassLoader 的父类。

热修补技术是怎样实现的，和插件化有什么区别？

插件化：动态加载主要解决 3 个技术问题：

- 1、使用 ClassLoader 加载类。
- 2、资源访问。
- 3、生命周期管理。

插件化是体现在功能拆分方面的，它将某个功能独立提取出来，独立开发，独立测试，再插入到主应用中。以此来减少主应用的规模。

热修复：

原因：因为一个 dvm 中存储方法 id 用的是 short 类型，导致 dex 中方法不能超过 65536 个。

代码热修复原理：

- 将编译好的 class 文件拆分打包成两个 dex，绕过 dex 方法数量的限制以及安装时的检查，在运行时再动态加载第二个 dex 文件中。
- 热修复是体现在 bug 修复方面的，它实现的是不需要重新发版和重新安装，就可以去修复已知的 bug。

- 利用 PathClassLoader 和 DexClassLoader 去加载与 bug 类同名的类，替换掉 bug 类，进而达到修复 bug 的目的，原理是在 app 打包的时候阻止类打上 CLASS_ISPREVERIFIED 标志，然后在热修复的时候动态改变 BaseDexClassLoader 对象间接引用的 dexElements，替换掉旧的类。

相同点:

都使用 ClassLoader 来实现加载新的功能类，都可以使用 PathClassLoader 与 DexClassLoader。

不同点：

热修复因为是为了修复 Bug 的，所以要将新的类替代同名的 Bug 类，要抢先加载新的类而不是 Bug 类，所以多做两件事：在原先的 app 打包的时候，阻止相关类去打上 CLASS_ISPREVERIFIED 标志，还有在热修复时动态改变 BaseDexClassLoader 对象间接引用的 dexElements，这样才能抢先代替 Bug 类，完成系统不加载旧的 Bug 类.。 而插件化只是增加新的功能类或者是资源文件，所以不涉及抢先加载新的类这样的使命，就避过了阻止相关类去打上 CLASS_ISPREVERIFIED 标志和还有在热修复时动态改变 BaseDexClassLoader 对象间接引用的 dexElements.

所以插件化比热修复简单，热修复是在插件化的基础上在进行替换旧的 Bug 类。

热修复原理：

资源修复：

很多热修复框架的资源修复参考了 Instant Run 的资源修复的原理。

传统编译部署流程如下：

Instant Run 编译部署流程如下：

- Hot Swap：修改一个现有方法中的代码时会采用 Hot Swap。
- Warm Swap：修改或删除一个现有的资源文件时会采用 Warm Swap。
- Cold Swap：有很多情况，如添加、删除或修改一个字段和方法、添加一个类等。

Instant Run 中的资源热修复流程：

- 1、创建新的 AssetManager，通过反射调用 addAssetPath 方法加载外部的资源，这样新创建的 AssetManager 就含有了外部资源。
- 2、将 AssetManager 类型的 mAssets 字段的引用全部替换为新创建的 AssetManager。

代码修复：

1、类加载方案：

65536 限制：

65536 的主要原因是 DVM Bytecode 的限制，DVM 指令集的方法调用指令 invoke-kind 索引为 16bits，最多能引用 65535 个方法。

LinearAlloc 限制：

- DVM 中的 LinearAlloc 是一个固定的缓存区，当方法数超过了缓存区的大小时会报错。

Dex 分包方案主要做的是在打包时将应用代码分成多个 Dex，将应用启动时必须用到的类和这些类的直接引用类放到 Dex 中，其他代码放到次 Dex 中。当应用启动时先加载主 Dex，等到应用启动后再动态地加载次 Dex，从而缓解了主 Dex 的 65536 限制和 LinearAlloc 限制。

加载流程：

- 根据 dex 文件的查找流程，我们将有 Bug 的类 Key.class 进行修改，再将 Key.class 打包成包含 dex 的补丁包 Patch.jar，放在 Element 数组 dexElements 的第一个元素，这样会首先找到 Patch.dex 中的 Key.class 去替换之前存在 Bug 的 Key.class，排在数组后面的 dex 文件中存在 Bug 的 Key.class 根据 ClassLoader 的双亲委托模式就不会被加载。

类加载方案需要重启 App 后让 ClassLoader 重新加载新的类，为什么需要重启呢？

- 这是因为类是无法被卸载的，要想重新加载新的类就需要重启 App，因此采用类加载方案的热修复框架是不能即时生效的。

各个热修复框架的实现细节差异：

- QQ 空间的超级补丁和 Nuwa 是按照上面说的将补丁包放在 Element 数组的第一个元素得到优先加载。
- 微信的 Tinker 将新旧 APK 做了 diff，得到 path.dex，再将 patch.dex 与手机中 APK 的 classes.dex 做合并，生成新的 classes.dex，然后在运行时通过反射将 classes.dex 放在 Elements 数组的第一个元素。

- 饿了么的 Amigo 则是将补丁包中每个 dex 对应的 Elements 取出来，之后组成新的 Element 数组，在运行时通过反射用新的 Elements 数组替换掉现有的 Elements 数组。

2、底层替换方案：

当我们要反射 Key 的 show 方法，会调用 Key.class.getDeclaredMethod("show").invoke(Key.class.newInstance());，最终会在 native 层将传入的 javaMethod 在 ART 虚拟机中对应一个 ArtMethod 指针，ArtMethod 结构体中包含了 Java 方法的所有信息，包括执行入口、访问权限、所属类和代码执行地址等。

替换 ArtMethod 结构体中的字段或者替换整个 ArtMethod 结构体，这就是底层替换方案。

AndFix 采用的是替换 ArtMethod 结构体中的字段，这样会有兼容性问题，因为厂商可能会修改 ArtMethod 结构体，导致方法替换失败。

Sophix 采用的是替换整个 ArtMethod 结构体，这样不会存在兼容问题。

底层替换方案直接替换了方法，可以立即生效不需要重启。采用底层替换方案主要是阿里系为主，包括 AndFix、Dexposed、阿里百川、Sophix。

3、Instant Run 方案：

什么是 ASM？

ASM 是一个 java 字节码操控框架，它能够动态生成类或者增强现有类的功能。ASM 可以直接产生 class 文件，也可以在类被加载到虚拟机之前动态改变类的行为。

Instant Run 在第一次构建 APK 时，使用 ASM 在每一个方法中注入了类似的代码逻辑：当$change 不为 null 时，则调用它的 access$dispatch 方法，参数为具体的方法名和方法参数。当 MainActivity 的 onCreate 方法做了修改，就会生成替换类 MainActivity$override，这个类实现了 IncrementalChange 接口，同时也会生成一个 AppPatchesLoaderImpl 类，这个类的 getPatchedClasses 方法会返回被修改的类的列表（里面包含了 MainActivity），根据列表会将 MainActivity 的$change 设置为 MainActivity$override。最后这个$change 就不会为 null，则会执行 MainActivity$override 的 access$dispatch 方法，最终会执行 onCreate 方法，从而实现了 onCreate 方法的修改。

借鉴 Instant Run 原理的热修复框架有 Robust 和 Aceso。

动态链接库修复：

重新加载 so。

加载 so 主要用到了 System 类的 load 和 loadLibrary 方法，最终都会调用到 nativeLoad 方法。其会调用 JavaVMExt 的 LoadNativeLibrary 函数来加载 so。

so 修复主要有两个方案：

- 1、将 so 补丁插入到 NativeLibraryElement 数组的前部，让 so 补丁的路径先被返回和加载。
- 2、调用 System 的 load 方法来接管 so 的加载入口。

为什么选用插件化？

在 Android 传统开发中，一旦应用的代码被打包成 APK 并被上传到各个应用市场，我们就不能修改应用的源码了，只能通过服务器来控制应用中预留的分支代码。但是很多时候我们无法预知需求和突然发生的情况，也就不能提前在应用代码中预留分支代码，这时就需要采用动态加载技术，即在程序运行时，动态加载一些程序中原本不存在的可执行文件并运行这些文件里的代码逻辑。其中可执行文件包括动态链接库 so 和 dex 相关文件（dex 以及包含 dex 的 jar/apk 文件）。随着应用开发技术和业务的逐步发展，动态加载技术派生出两个技术：热修复和插件化。其中热修复技术主要用来修复 Bug，而插件化技术则主要用于解决应用越来越庞大以及功能模块的解耦。详细点说，就是为了解决以下几种情况：

- 1、业务复杂、模块耦合：随着业务越来越复杂，应用程序的工程和功能模块数量会越来越多，一个应用可能由几十甚至几百人来协同开发，其中的一个工程可能就由一个小组来进行开发维护，如果功能模块间的耦合度较高，修改一个模块会影响其它功能模块，势必会极大地增加沟通成本。
- 2、应用间的接入：当一个应用需要接入其它应用时，如淘宝，为了将流量引流到其它的淘宝应用如：飞猪旅游、口碑外卖、聚划算等等应用，如使用常规技术有两个问题：可能要维护多个版本的问题或单个应用体积将会非常庞大的问题。
- 3、65536 限制，内存占用大。

插件化的思想：

安装的应用可以理解为插件，这些插件可以自由地进行插拔。

插件化的定义：

插件一般是指经过处理的 APK，so 和 dex 等文件，插件可以被宿主进行加载，

有的插件也可以作为 APK 独立运行。

将一个应用按照插件的方式进行改造的过程就叫作插件化。

插件化的优势：

- 低耦合
- 应用间的接入和维护更便捷，每个应用团队只需要负责自己的那一部分。

- 应用及主 dex 的体积也会相应变小，间接地避免了 65536 限制。
- 第一次加载到内存的只有淘宝客户端，当使用到其它插件时才会加载相应插件到内存，以减少内存占用。

插件化框架对比：

- 最早的插件化框架：2012 年大众点评的屠毅敏就推出了 AndroidDynamicLoader 框架。
- 目前主流的插件化方案有滴滴任玉刚的 VirtualApk、360 的 DroidPlugin、RePlugin、Wequick 的 Small 框架。
- 如果加载的插件不需要和宿主有任何耦合，也无须和宿主进行通信，比如加载第三方 App，那么推荐使用 RePlugin，其他情况推荐使用 VirtualApk。

  由于 VirtualApk 在加载耦合插件方面是插件化框架的首选，具有普遍的适用性，因此有必要对它的源码进行了解。

插件化原理：

Activity 插件化：

主要实现方式有三种：

- 反射：对性能有影响，主流的插件化框架没有采用此方式。
- 接口：dynamic-load-apk 采用。
- Hook：主流。

Hook 实现方式有两种：Hook IActivityManager 和 Hook Instrumentation。主要方案就是先用一个在 AndroidManifest.xml 中注册的 Activity 来进行占坑，用来通过 AMS 的校验，接着在合适的时机用插件 Activity 替换占坑的 Activity。

Hook IActivityManager：

1、占坑、通过校验：

在 Android 7.0 和 8.0 的源码中 IActivityManager 借助了 Singleton 类实现单例，而且该单例是静态的，因此 IActivityManager 是一个比较好的 Hook 点。

接着，定义替换 IActivityManager 的代理类 IActivityManagerProxy，由于 Hook 点 IActivityManager 是一个接口，建议这里采用动态代理。

- 拦截 startActivity 方法，获取参数 args 中保存的 Intent 对象，它是原本要启动插件 TargetActivity 的 Intent。
- 新建一个 subIntent 用来启动 StubActivity，并将前面得到的 TargetActivity 的 Intent 保存到 subIntent 中，便于以后还原 TargetActivity。
- 最后，将 subIntent 赋值给参数 args，这样启动的目标就变为了 StubActivity，用来通过 AMS 的校验。

然后，用代理类 IActivityManagerProxy 来替换 IActivityManager。

- 当版本大于等于 26 时，使用反射获取 ActivityManager 的 IActivityManagerSingleton 字段，小于时则获取 ActivityManagerNative 中的 gDefault 字段。
- 然后，通过反射获取对应的 Singleton 实例，从上面得到的 2 个字段中拿到对应的 IActivityManager。
- 最后，使用 Proxy.newProxyInstance() 方法动态创建代理类 IActivityManagerProxy，用 IActivityManagerProxy 来替换 IActivityManager。

2、还原插件 Activity：

- 前面用占坑 Activity 通过了 AMS 的校验，但是我们要启动的是插件 TargetActivity，还需要用插件 TargetActivity 来替换占坑的 SubActivity，替换时机为图中步骤 2 之后。
- 在 ActivityThread 的 H 类中重写的 handleMessage 方法会对 LAUNCH_ACTIVITY 类型的消息进行处理，最终会调用 Activity 的 onCreate 方法。在 Handler 的 dispatchMessage 处理消息的这个方法中，看到如果 Handelr 的 Callback 类型的 mCallBack 不为 null，就会执行 mCallback 的 handleMessage 方法，因此 mCallback 可以作为 Hook 点。我们可以用自定义的 Callback 来替换 mCallback。

自定义的 Callback 实现了 Handler.Callback，并重写了 handleMessage 方法，当收到消息的类型为 LAUNCH_ACTIVITY 时，将启动 SubActivity 的 Intent 替换为启动 TargetActivity 的 Intent。然后使用反射将 Handler 的 mCallback 替换为自定义的 CallBack 即可。使用时则在 application 的 attachBaseContext 方法中进行 hook 即可。

3、插件 Activity 的生命周期：

- AMS 和 ActivityThread 之间的通信采用了 token 来对 Activity 进行标识，并且此后的 Activity 的生命周期处理也是根据 token 来对 Activity 进行标识的，因为我们在 Activity 启动时用插件 TargetActivity 替换占坑 SubActivity，这一过程在 performLaunchActivity 之前，因此

performLaunchActivity 的 r.token 就是 TargetActivity。所以 TargetActivity 具有生命周期。

Hook Instrumentation：

Hook Instrumentation 实现同样也需要用到占坑 Activity，与 Hook IActivity 实现不同的是，用占坑 Activity 替换插件 Activity 以及还原插件 Activity 的地方不同。

分析：在 Activity 通过 AMS 校验前，会调用 Activity 的 startActivityForResult 方法，其中调用了 Instrumentation 的 execStartActivity 方法来激活 Activity 的生命周期。并且在 ActivityThread 的 performLaunchActivity 中使用了 mInstrumentation 的 newActivity 方法，其内部会用类加载器来创建 Activity 的实例。

方案：在 Instrumentation 的 execStartActivity 方法中用占坑 SubActivity 来通过 AMS 的验证，在 Instrumentation 的 newActivity 方法中还原 TargetActivity，这两部操作都和 Instrumentation 有关，因此我们可以用自定义的 Instumentation 来替换掉 mInstrumentation。具体为：

- 首先检查 TargetActivity 是否已经注册，如果没有则将 TargetActivity 的 ClassName 保存起来用于后面还原。接着把要启动的 TargetActivity 替换为 StubActivity，最后通过反射调用 execStartActivity 方法，这样就可以用 StubActivity 通过 AMS 的验证。
- 在 newActivity 方法中创建了此前保存的 TargetActivity，完成了还原 TargetActivity 。最后使用反射用 InstrumentationProxy 替换 mInstumentation。

资源插件化：

资源的插件化和热修复的资源修复都借助了 AssetManager。

资源的插件化方案主要有两种：

- 1、合并资源方案，将插件的资源全部添加到宿主的 Resources 中，这种方案插件可以访问宿主的资源。
- 2、构建插件资源方案，每个插件都构造出独立的 Resources，这种方案插件不可以访问宿主资源。

so 的插件化：

so 的插件化方案和 so 热修复的第一种方案类似，就是将 so 插件插入到 NativelibraryElement 数组中，并且将存储 so 插件的文件添加到 nativeLibraryDirectories 集合中就可以了。

插件的加载机制方案：

- 1、Hook ClassLoader。
- 2、委托给系统的 ClassLoader 帮忙加载。

2、模块化和组件化

模块化的好处

https://www.jianshu.com/p/376ea8a19a17

分析现有的组件化方案：

很多大厂的组件化方案是以 多工程 + 多 Module 的结构(微信, 美团等超级 App 更是以 多工程 + 多 Module + 多 P 工程(以页面为单元的代码隔离方式) 的三级工程结构), 使用 Git Submodule 创建多个子仓库管理各个模块的代

码，并将各个模块的代码打包成 AAR 上传至私有 Maven 仓库使用远程版本号依赖的方式进行模块间代码的隔离。

组件化开发的好处：

- 避免重复造轮子，可以节省开发和维护的成本。
- 可以通过组件和模块为业务基准合理地安排人力，提高开发效率。
- 不同的项目可以共用一个组件或模块，确保整体技术方案的统一性。
- 为未来插件化共用同一套底层模型做准备。

跨组件通信：

跨组件通信场景：

- 第一种是组件之间的页面跳转 (Activity 到 Activity, Fragment 到 Fragment, Activity 到 Fragment, Fragment 到 Activity) 以及跳转时的数据传递 (基础数据类型和可序列化的自定义类类型)。
- 第二种是组件之间的自定义类和自定义方法的调用(组件向外提供服务)。

跨组件通信方案分析：

- 第一种组件之间的页面跳转不需要过多描述了，算是 ARouter 中最基础的功能, API 也比较简单，跳转时想传递不同类型的数据也提供有相应的 API。
- 第二种组件之间的自定义类和自定义方法的调用要稍微复杂点，需要 ARouter 配合架构中的 公共服务(CommonService) 实现：

提供服务的业务模块：

在公共服务(CommonService) 中声明 Service 接口 (含有需要被调用的自定义方法)，然后在自己的模块中实现这个 Service 接口，再通过 ARouter API 暴露实现类。

使用服务的业务模块：

通过 ARouter 的 API 拿到这个 Service 接口(多态持有，实际持有实现类)，即可调用 Service 接口中声明的自定义方法，这样就可以达到模块之间的交互。此外，可以使用 AndroidEventBus 其独有的 Tag，可以在开发时更容易定位发送事件和接受事件的代码，如果以组件名来作为 Tag 的前缀进行分组，也可以更好的统一管理和查看每个组件的事件，当然也不建议大家过多使用 EventBus。

如何管理过多的路由表？

RouterHub 存在于基础库，可以被看作是所有组件都需要遵守的通讯协议，里面不仅可以放路由地址常量，还可以放跨组件传递数据时命名的各种 Key 值，再配以适当注释，任何组件开发人员不需要事先沟通只要依赖了这个协议，就知道了各自该怎样协同工作，既提高了效率又降低了出错风险，约定的东西自然要比口头上说的强。

Tips: 如果您觉得把每个路由地址都写在基础库的 RouterHub 中，太麻烦了，也可以在每个组件内部建立一个私有 RouterHub，将不需要跨组件的路由地址放入私有 RouterHub 中管理，只将需要跨组件的路由地址放入基础库的公有 RouterHub 中管理，如果您不需要集中管理所有路由地址的话，这也是比较推荐的一种方式。

ARouter 路由原理：

ARouter 维护了一个路由表 Warehouse，其中保存着全部的模块跳转关系，ARouter 路由跳转实际上还是调用了 startActivity 的跳转，使用了原生的 Framework 机制，只是通过 apt 注解的形式制造出跳转规则，并人为地拦截跳转和设置跳转条件。

多模块开发的时候不同的负责人可能会引入重复资源，相同的字符串，相同的 icon 等但是文件名并不一样，怎样去重？

3、gradle

gradle 熟悉么，自动打包知道么？

如何加快 Gradle 的编译速度？

Gradle 的 Flavor 能否配置 sourceset？

Gradle 生命周期

4、编译插桩

谈谈你对 AOP 技术的理解？

说说你了解的编译插桩技术？

五、架构设计

MVC MVP MVVM 原理和区别？

架构设计的目的

通过设计是模块程序化，从而做到高内聚低耦合，让开发者能更专注于功能实现本身，提供程序开发效率、更容易进行测试、维护和定位问题等等。而且，不同的规模的项目应该选用不同的架构设计。

MVC

MVC 是模型(model)－视图(view)－控制器(controller)的缩写，其中 M 层处理数据，业务逻辑等；V 层处理界面的显示结果；C 层起到桥梁的作用，来控制 V 层和 M 层通信以此来达到分离视图显示和业务逻辑层。在 Android 中的 MVC 划分是这样的：

- 视图层(View)：一般采用 XML 文件进行界面的描述，也可以在界面中使用动态布局的方式。

- 控制层(Controller)：由 Activity 承担。
- 模型层(Model)：数据库的操作、对网络等的操作，复杂业务计算等等。

MVC 缺点

在 Android 开发中，Activity 并不是一个标准的 MVC 模式中的 Controller，它的首要职责是加载应用的布局和初始化用户界面，并接受和处理来自用户的操作请求，进而作出响应。随着界面及其逻辑的复杂度不断提升，Activity 类的职责不断增加，以致变得庞大臃肿。

MVP

MVP 框架由 3 部分组成：View 负责显示，Presenter 负责逻辑处理，Model 提供数据。

- View:负责绘制 UI 元素、与用户进行交互(在 Android 中体现为 Activity)。
- Model:负责存储、检索、操纵数据(有时也实现一个 Model interface 用来降低耦合)。
- Presenter:作为 View 与 Model 交互的中间纽带，处理与用户交互的逻辑。
- View interface:需要 View 实现的接口，View 通过 View interface 与 Presenter 进行交互，降低耦合，方便使用 MOCK 对 Presenter 进行单元测试。

MVP 的 Presenter 是框架的控制者，承担了大量的逻辑操作，而 MVC 的 Controller 更多时候承担一种转发的作用。因此在 App 中引入 MVP 的原因，是为了将此前在 Activty 中包含的大量逻辑操作放到控制层中，避免 Activity 的臃肿。

MVP 与 MVC 的主要区别:

- 1、（最主要区别）View 与 Model 并不直接交互，而是通过与 Presenter 交互来与 Model 间接交互。而在 MVC 中 View 可以与 Model 直接交互。
- 2、Presenter 与 View 的交互是通过接口来进行的，更有利于添加单元测试。

MVP 的优点

- 1、模型与视图完全分离，我们可以修改视图而不影响模型。
- 2、可以更高效地使用模型，因为所有的交互都发生在一个地方——Presenter 内部。
- 3、我们可以将一个 Presenter 用于多个视图，而不需要改变 Presenter 的逻辑。这个特性非常的有用，因为视图的变化总是比模型的变化频繁。
- 4、如果我们把逻辑放在 Presenter 中，那么我们就可以脱离用户接口来测试这些逻辑（单元测试）。

UI层一般包括Activity，Fragment，Adapter等直接和UI相关的类，UI层的Activity在启动之后实例化相应的Presenter，App的控制权后移，由UI转移到Presenter，两者之间的通信通过BroadCast、Handler、事件总线机制或者接口完成，只传递事件和结果。

MVP 的执行流程：首先 V 层通知 P 层用户发起了一个网络请求，P 层会决定使用负责网络相关的 M 层去发起请求网络，最后，P 层将完成的结果更新到 V 层。

MVP 的变种：Passive View

View 直接依赖 Presenter，但是 Presenter 间接依赖 View，它直接依赖的是 View 实现的接口。相对于 View 的被动，那 Presenter 就是主动的一方。对于 Presenter 的主动，有如下的理解：

- Presenter 是整个 MVP 体系的控制中心，而不是单纯的处理 View 请求的人。
- View 仅仅是用户交互请求的汇报者，对于响应用户交互相关的逻辑和流程，View 不参与决策，真正的决策者是 Presenter。
- View 向 Presenter 发送用户交互请求应该采用这样的口吻：“我现在将用户交互请求发送给你，你看着办，需要我的时候我会协助你”。
- 对于绑定到 View 上的数据，不应该是 View 从 Presenter 上“拉”回来的，应该是 Presenter 主动“推”给 View 的。（这里借鉴了 IOC 做法）
- View 尽可能不维护数据状态，因为其本身仅仅实现单纯的、独立的 UI 操作；Presenter 才是整个体系的协调者，它根据处理用于交互的逻辑给 View 和 Model 安排工作。

MVP 架构存在的问题与解决办法

- 1、加入模板方法

将逻辑操作从 V 层转移到 P 层后，可能有一些 Activity 还是比较膨胀，此时，可以通过继承 BaseActivity 的方式加入模板方法。注意，最好不要超过 3 层继承。

- 2、Model 内部分层

模型层（Model）中的整体代码量是最大的，此时可以进行模块的划分和接口隔离。

- 3、使用中介者和代理

在 UI 层和 Presenter 之间设置中介者 Mediator，将例如数据校验、组装在内的轻量级逻辑操作放在 Mediator 中；在 Presenter 和 Model 之间使用代理 Proxy；通过上述两者分担一部分 Presenter 的逻辑操作，但整体框架的控制权还是在 Presenter 手中。

MVVM

MVVM 可以算是 MVP 的升级版，其中的 VM 是 ViewModel 的缩写，ViewModel 可以理解成是 View 的数据模型和 Presenter 的合体，ViewModel 和 View 之间的交互通过 Data Binding 完成，而 Data Binding 可以实现双向的交互，这就使得视图和控制层之间的耦合程度进一步降低，关注点分离更为彻底，同时减轻了 Activity 的压力。

MVC->MVP->MVVM 演进过程

MVC -> MVP -> MVVM 这几个软件设计模式是一步步演化发展的，MVVM 是从 MVP 的进一步发展与规范，MVP 隔离了 MVC 中的 M 与 V 的直接联系后，靠 Presenter 来中转，所以使用 MVP 时 P 是直接调用 View 的接口来实现对视图的操作的，这个 View 接口的东西一般来说是 showData、showLoading 等等。M 与 V 已经隔离了，方便测试了，但代码还不够优雅简洁，所以 MVVM 就弥补了这些缺陷。在 MVVM 中就出现的 Data Binding 这个概念，意思就是 View 接口的 showData 这些实现方法可以不写了，通过 Binding 来实现。

三种模式的相同点

M 层和 V 层的实现是一样的。

三种模式的不同点

三者的差异在于如何粘合 View 和 Model，实现用户的交互操作以及变更通知。

- Controller：接收 View 的命令，对 Model 进行操作，一个 Controller 可以对应多个 View。
- Presenter：Presenter 与 Controller 一样，接收 View 的命令，对 Model 进行操作；与 Controller 不同的是 Presenter 会反作用于 View，Model 的变更通知首先被 Presenter 获得，然后 Presenter 再去更新 View。通常一个 Presenter 只对应于一个 View。据 Presenter 和 View 对逻辑代码分担的程度不同，这种模式又有两种情况：普通的 MVP 模式和 Passive View 模式。
- ViewModel：注意这里的“Model”指的是 View 的 Model，跟 MVVM 中的一个 Model 不是一回事。所谓 View 的 Model 就是包含 View 的一些数据属性和操作的这么一个东东，这种模式的关键技术就是数据绑定（data binding），View 的变化会直接影响 ViewModel，ViewModel 的变化或者内容也会直接体现在 View 上。这种模式实际上是框架替应用开发者做了一些工作，开发者只需要较少的代码就能实现比较复杂的交互。

补充：基于 AOP 的架构设计

AOP(Aspect-Oriented Programming, 面向切面编程)，诞生于上个世纪 90 年代，是对 OOP(Object-Oriented Programming, 面向对象编程)的补充和完善。OOP 引入封装、继承和多态性等概念来建立一种从上道下的对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，即定义从左到右的关系时，OOP 则显得无能为力。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（Cross-Cutting）代码，在 OOP 设计中，它导致了大量

代码的重复，而不利于各个模块的重用。而 AOP 技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

在 Android App 中的横切关注点有 Http, SharedPreferences, Log, Json, Xml, File, Device, System, 格式转换等。Android App 的需求差别很大，不同的需求横切关注点必然是不一样的。一般的 App 工程中应该有一个 Util Package 来存放相关的切面操作，在项目多了之后可以将其中使用较多的 Util 封装为一个 Jar 包/aar 文件/远程依赖的方式供工程调用。

在使用 MVP 和 AOP 对 App 进行纵向和横向的切割之后，能够使得 App 整体的结构更清晰合理，避免局部的代码臃肿，方便开发、测试以及后续的维护。这样纵，横两次对于 App 代码的分割已经能使得程序不会过多堆积在一个 Java 文件里，但靠一次开发过程就写出高质量的代码是很困难的，趁着项目的间歇期，对代码进行重构很有必要。

最后的建议

如果“从零开始”，用什么设计架构的问题属于想得太多做得太少的问题。 从零开始意味着一个项目的主要技术难点是基本功能实现。当每一个功能都需要考虑如何做到的时候，我觉得一般人都没办法考虑如何做好。 因为，所有的优化都是站在最上层进行统筹规划。在这之前，你必须对下层的每一个模块都非常熟悉，进而提炼可复用的代码、规划逻辑流程。

MVC 的情况下怎么把 Activity 的 C 和 V 抽离？

MVP 架构中 Presenter 定义为接口有什么好处；

MVP 如何管理 Presenter 的生命周期，何时取消网络请求？

aop 思想

Fragment 如果在 Adapter 中使用应该如何解耦？

项目框架里有没有 Base 类，BaseActivity 和 BaseFragment 这种封装导致的问题，以及解决方法？

设计一个音乐播放界面，你会如何实现，用到那些类，如何设计，如何定义接口，如何与后台交互，如何缓存与下载，如何优化(15 分钟时间)

从 0 设计一款 App 整体架构，如何去做？

说一款你认为当前比较火的应用并设计(比如：直播 APP，P2P 金融，小视频等)

实现一个库，完成日志的实时上报和延迟上报两种功能，该从哪些方面考虑？

你最优秀的工程设计项目，是怎么设计和实现的；扩展，如何做成一个平台级产品？

六、其它高频面试题

1、如何保证一个后台服务不被杀死？（相同问题：如何保证 service 在后台不被 kill？）比较省电的方式是什么？

保活方案

1、AIDL 方式单进程、双进程方式保活 Service。（基于 onStartCommand() return START_STICKY）

START_STICKY 在运行 onStartCommand 后 service 进程被 kill 后，那将保留在开始状态，但是不保留那些传入的 intent。不久后 service 就会再次尝试重新创

建，因为保留在开始状态，在创建 service 后将保证调用 onstartCommand。如果没有传递任何开始命令给 service，那将获取到 null 的 intent。

除了华为此方案无效以及未更改底层的厂商不起作用外（START_STICKY 字段就可以保持 Service 不被杀）。此方案可以与其他方案混合使用

2、降低 oom_adj 的值（提升 service 进程优先级）：

Android 中的进程是托管的，当系统进程空间紧张的时候，会依照优先级自动进行进程的回收。Android 将进程分为 6 个等级,它们按优先级顺序由高到低依次是:

- 1.前台进程 (Foreground process)
- 2.可见进程 (Visible process)
- 3.服务进程 (Service process)
- 4.后台进程 (Background process)
- 5.空进程 (Empty process)

当 service 运行在低内存的环境时，将会 kill 掉一些存在的进程。因此进程的优先级将会很重要，可以使用 startForeground 将 service 放到前台状态。这样在低内存时被 kill 的几率会低一些。

> 常驻通知栏（可通过启动另外一个服务关闭 Notification，不对 oom_adj 值有影响）。

> 使用”1 像素“的 Activity 覆盖在 getWindow()的 view 上。

此方案无效果

- 循环播放无声音频（黑科技，7.0 下杀不掉）。

成功对华为手机保活。小米 8 下也成功突破 20 分钟

- 3、监听锁屏广播：使 Activity 始终保持前台。
- 4、使用自定义锁屏界面：覆盖了系统锁屏界面。
- 5、通过 android:process 属性来为 Service 创建一个进程。
- 6、跳转到系统白名单界面让用户自己添加 app 进入白名单。

复活方案

1、onDestroy 方法里重启 service

service + broadcast 方式，就是当 service 走 onDestory 的时候，发送一个自定义的广播，当收到广播的时候，重新启动 service。

2、JobScheduler：原理类似定时器，5.0,5.1,6.0 作用很大，7.0 时候有一定影响（可以在电源管理中给 APP 授权）。

只对 5.0，5.1、6.0 起作用。

3、推送互相唤醒复活：极光、友盟、以及各大厂商的推送。

4、同派系 APP 广播互相唤醒：比如今日头条系、阿里系。

此外还可以监听系统广播判断 Service 状态，通过系统的一些广播，比如：手机重启、界面唤醒、应用状态改变等等监听并捕获到，然后判断我们的 Service 是否还存活。

结论：高版本情况下可以使用弹出通知栏、双进程、无声音乐提高后台服务的保活概率。

2、Android 动画框架实现原理。

Animation 框架定义了透明度，旋转，缩放和位移几种常见的动画，而且控制的是整个 View。实现原理：

每次绘制视图时，View 所在的 ViewGroup 中的 drawChild 函数获取该 View 的 Animation 的 Transformation 值，然后调用 canvas.concat(transformToApply.getMatrix())，通过矩阵运算完成动画帧，如果动画没有完成，继续调用 invalidate() 函数，启动下次绘制来驱动动画，动画过程中的帧之间间隙时间是绘制函数所消耗的时间，可能会导致动画消耗比较多的 CPU 资源，最重要的是，动画改变的只是显示，并不能响应事件。

3、Activity-Window-View 三者的差别？

Activity 像一个工匠（控制单元），Window 像窗户（承载模型），View 像窗花（显示视图） LayoutInflater 像剪刀，Xml 配置像窗花图纸。

在 Activity 中调用 attach，创建了一个 Window， 创建的 window 是其子类 PhoneWindow，在 attach 中创建 PhoneWindow。 在 Activity 中调用 setContentView(R.layout.xxx)， 其中实际上是调用的 getWindow().setContentView()， 内部调用了 PhoneWindow 中的 setContentView 方法。

创建 ParentView：

作为 ViewGroup 的子类，实际是创建的 DecorView(作为 FramLayout 的子类），将指定的 R.layout.xxx 进行填充， 通过布局填充器进行填充【其中的 parent 指的就是 DecorView】， 调用 ViewGroup 的 removeAllView()，先将所有的 view 移除掉，添加新的 view：addView()。

参考文章

4、低版本 SDK 如何实现高版本 api？

- 1、在使用了高版本 API 的方法前面加一个 @TargetApi(API 号)。
- 2、在代码上用版本判断来控制不同版本使用不同的代码。

5、说说你对 Context 的理解？

6、Android 的生命周期和启动模式

由 A 启动 B Activity，A 为栈内复用模式，B 为标准模式，然后再次启动 A 或者杀死 B，说说 A，B 的生命周期变化，为什么？

Activity 的启动模式有哪些？栈里是 A-B-C，先想直接到 A，BC 都清理掉，有几种方法可以做到？这几种方法产生的结果是有几个 A 的实例？

7、ListView 和 RecyclerView 系列

RecyclerView 和 ListView 有什么区别？局部刷新？前者使用时多重 type 场景下怎么避免滑动卡顿。懒加载怎么实现，怎么优化滑动体验。

ListView、RecyclerView 区别？

一、使用方面：

ListView 的基础使用：

- 继承重写 BaseAdapter 类

- 自定义 ViewHolder 和 convertView 一起完成复用优化工作

RecyclerView 基础使用关键点同样有两点：

- 继承重写 RecyclerView.Adapter 和 RecyclerView.ViewHolder
- 设置布局管理器，控制布局效果

RecyclerView 相比 ListView 在基础使用上的区别主要有如下几点：

- ViewHolder 的编写规范化了
- RecyclerView 复用 Item 的工作 Google 全帮你搞定，不再需要像 ListView 那样自己调用 setTag
- RecyclerView 需要多出一步 LayoutManager 的设置工作

二、布局方面：

RecyclerView 支持 线性布局、网格布局、瀑布流布局 三种，而且同时还能够控制横向还是纵向滚动。

三、API 提供方面：

ListView 提供了 setEmptyView ，addFooterView 、 addHeaderView.

RecyclerView 供了 notifyItemChanged 用于更新单个 Item View 的刷新，我们可以省去自己写局部更新的工作。

四、动画效果：

RecyclerView 在做局部刷新的时候有一个渐变的动画效果。继承

RecyclerView.ItemAnimator 类，并实现相应的方法，再调用 RecyclerView 的

setItemAnimator(RecyclerView.ItemAnimator animator) 方法设置完即可实现自定义的动画效果。

五、监听 Item 的事件：

ListView 提供了单击、长按、选中某个 Item 的监听设置。

RecyclerView 与 ListView 缓存机制的不同

想改变 listview 的高度，怎么做？

listview 跟 recyclerview 上拉加载的时候分别应该如何处理？

如何自己实现 RecyclerView 的侧滑删除？

RecyclerView 的 ItemTouchHelper 的实现原理

8、如何实现一个推送，消息推送原理？推送到达率的问题？

一：客户端不断的查询服务器，检索新内容，也就是所谓的 pull 或者轮询方式。

二：客户端和服务器之间维持一个 TCP/IP 长连接，服务器向客户端 push。

https://blog.csdn.net/clh604/article/details/20167263

https://www.jianshu.com/p/45202dcd5688

9、动态权限系列。

动态权限适配方案，权限组的概念

Runtime permission，如何把一个预置的 app 默认给它权限？不要授权。

10、自定义 View 系列。

Canvas 的底层机制，绘制框架，硬件加速是什么原理，canvas lock 的缓冲区是怎么回事？

双指缩放拖动大图
TabLayout 中如何让当前标签永远位于屏幕中间
TabLayout 如何设置指示器的宽度包裹内容？
自定义 View 如何考虑机型适配？

- 合理使用 warp_content，match_parent。
- 尽可能地使用 RelativeLayout。
- 针对不同的机型，使用不同的布局文件放在对应的目录下，android 会自动匹配。
- 尽量使用点 9 图片。
- 使用与密度无关的像素单位 dp，sp。
- 引入 android 的百分比布局。
- 切图的时候切大分辨率的图，应用到布局当中，在小分辨率的手机上也会有很好的显示效果。

11、对谷歌新推出的 Room 架构。

12、没有给权限如何定位，特定机型定位失败，如何解决?

13、Debug 跟 Release 的 APK 的区别？

14、android 文件存储，各版本存储位置的权限控制的演进，外部存储，内部存储

15、有什么提高编译速度的方法？

16、Scroller 原理。

Scroller 执行流程里面的三个核心方法

```
mScroller.startScroll();

mScroller.computeScrollOffset();
```

```
view.computeScroll();
```

1、在 mScroller.startScroll()中为滑动做了一些初始化准备，比如：起始坐标，滑动的距离和方向以及持续时间(有默认值)，动画开始时间等。

2、mScroller.computeScrollOffset()方法主要是根据当前已经消逝的时间来计算当前的坐标点。因为在 mScroller.startScroll()中设置了动画时间，那么在 computeScrollOffset()方法中依据已经消逝的时间就很容易得到当前时刻应该所处的位置并将其保存在变量 mCurrX 和 mCurrY 中。除此之外该方法还可判断动画是否已经结束。

17、Hybrid 系列。

webwiew 了解？怎么实现和 javascript 的通信？相互双方的通信。@JavascriptInterface 在？版本有 bug，除了这个还有其他调用 android 方法的方案吗？

Android 中 Java 和 JavaScript 交互

```
webView.addJavaScriptInterface(new Object(){xxx}, "xxx");

1
```

答案：可以使用 WebView 控件执行 JavaScript 脚本，并且可以在 JavaScript 中执行 Java 代码。要想让 WebView 控件执行 JavaScript，需要调用 WebSettings.setJavaScriptEnabled 方法，代码如下：

```
WebView webView = (WebView)findViewById(R.id.webview);

WebSettings webSettings = webView.getSettings();

//设置 WebView 支持 JavaScript

webSettings.setJavaScriptEnabled(true);

webView.setWebChromeClient(new WebChromeClient());
```

JavaScript 调用 Java 方法需要使用 WebView.addJavascriptInterface 方法设置 JavaScript 调用的 Java 方法，代码如下：

```
webView.addJavascriptInterface(new Object()
{
    //JavaScript 调用的方法
    public String process(String value)
    {
        //处理代码
        return result;
    }
}, "demo");       //demo 是 Java 对象映射到 JavaScript 中的对象名
```

可以使用下面的 JavaScript 代码调用 process 方法，代码如下：

```
<script language="javascript">
    function search()
    {
        //调用 searchWord 方法
        result.innerHTML = "<font color='red'>" + window.demo.process('data')
+ "</font>";
    }
```

18、如果在当前线程内使用 Handler postdelayed 两个消息，一个延迟 5s，一个延迟 10s，然后使当前线程 sleep 5 秒，以上消息的执行时间会如何变化？

答：照常执行

扩展：sleep 时间<=5 对两个消息无影响，5< sleep 时间 <=10 对第一个消息有影响，第一个消息会延迟到 sleep 后执行，sleep 时间>10 对两个时间都有影响，都会延迟到 sleep 后执行。

19、Android 中进程内存的分配，能不能自己分配定额内存？

20、下拉状态栏是不是影响 activity 的生命周期，如果在 onStop 的时候做了网络请求，onResume 的时候怎么恢复

21、Android 长连接，怎么处理心跳机制。

长连接：长连接是建立连接之后，不主动断开. 双方互相发送数据，发完了也不主动断开连接，之后有需要发送的数据就继续通过这个连接发送.

心跳包：其实主要是为了防止 NAT 超时，客户端隔一段时间就主动发一个数据，探测连接是否断开。

服务器处理心跳包：假如客户端心跳间隔是固定的，那么服务器在连接闲置超过这个时间还没收到心跳时，可以认为对方掉线，关闭连接. 如果客户端心跳会动态改变，应当设置一个最大值，超过这个最大值才认为对方掉线. 还有一种情况就是服务器通过 TCP 连接主动给客户端发消息出现写超时，可以直接认为对方掉线.

22、CrashHandler 实现原理？

获取 app crash 的信息保存在本地然后在下一次打开 app 的时候发送到服务器。

23、SurfaceView 和 View 的最本质的区别？

SurfaceView 是在一个新起的单独线程中可以重新绘制画面，而 view 必须在 UI 的主线程中更新画面。

在 UI 的主线程中更新画面可能会引发问题，比如你更新的时间过长，那么你的主 UI 线程就会被你正在画的函数阻塞。那么将无法响应按键、触屏等消息。当使用 SurfaceView 由于是在新的线程中更新画面所以不会阻塞你的 UI 主线程。但这也带来了另外一个问题，就是事件同步。比如你触屏了一下，你需要在 SurfaceView 中的 thread 处理，一般就需要有一个 event queue 的设计来保存 touchevent，这会稍稍复杂一点，因为涉及到线程安全。

24、Android 程序运行时权限与文件系统权限

1、Linux 文件系统权限。不同的用户对文件有不同的读写执行权限。在 android 系统中，system 和应用程序是分开的，system 里的数据是不可更改的。

2、Android 中有 3 种权限，进程权限 UserID，签名，应用申明权限。每次安装时，系统根据包名为应用分配唯一的 userID，不同的 userID 运行在不同的进程里，进程间的内存是独立的，不可以相互访问，除非通过特定的 Binder 机制。

Android 提供了如下的一种机制，可以使两个 apk 打破前面讲的这种壁垒。

在 AndroidManifest.xml 中利用 sharedUserId 属性给不同的 package 分配相同的 userID，通过这样做，两个 package 可以被当做同一个程序，系统会分配给两个程序相同的 UserID。当然，基于安全考虑，两个 package 需要有相同的签名，否则没有验证也就没有意义了。

25、曲面屏的适配。

26、TextView 调用 setText 方法的内部执行流程。

27、怎么控制另外一个进程的 View 显示（RemoteView）？

28、如何实现右滑 finish activity?

29、如何在整个系统层面实现界面的圆角效果。（即所有的 APP 打开界面都会是圆角）

30、非 UI 线程可以更新 UI 吗?

可以，当访问 UI 时，ViewRootImpl 会调用 checkThread 方法去检查当前访问 UI 的线程是哪个，如果不是 UI 线程则会抛出异常。执行 onCreate 方法的那个时候 ViewRootImpl 还没创建，无法去检查当前线程.ViewRootImpl 的创建在 onResume 方法回调之后。

```
void checkThread() {

    if (mThread != Thread.currentThread()) {

        throw new CalledFromWrongThreadException(

                "Only the original thread that created a view hierarchy can touch
its views.");

    }

}
```

非 UI 线程是可以刷新 UI 的，前提是它要拥有自己的 ViewRoot,即更新 UI 的线程和创建 ViewRoot 的线程是同一个，或者在执行 checkThread()前更新 UI。

31、如何解决 git 冲突?

32、单元测试有没有做过，说说熟悉的单元测试框架?

首先，Android 测试主要分为三个方面：

- 单元测试（Junit4、Mockito、PowerMockito、Robolectric）
- UI 测试（Espresso、UI Automator）
- 压力测试（Monkey）

WanAndroid 项目和 XXX 项目中使用用到了单元测试和部分自动化 UI 测试，其中单元测试使用的是 Junit4+Mockito+PowerMockito+Robolectric。下面我分别简单介绍下这些测试框架：

1、Junit4：

使用@Test 注解指定一个方法为一个测试方法，除此之外，还有如下常用注解 @BeforeClass->@Before->@Test->@After->@AfterClass 以及@Ignore。

Junit4 的主要测试方法就是断言，即 assertEquals()方法。然后，你可以通过实现 TestRule 接口的方式重写 apply()方法去自定义 Junit Rule，这样就可以在执行测试方法的前后做一些通用的初始化或释放资源等工作，接着在想要的测试类中使用@Rule 注解声明使用 JsonChaoRule 即可。（注意被@Rule 注解的变量必须是 final 的。最后，我们直接运行对应的单元测试方法或类，如果你想要一键运行项目中所有的单元测试类，直接点击运行 Gradle Projects 下的 app/Tasks/verification/test 即可，它会在 module 下的 build/reports/tests/下生成对应的 index.html 报告。

Junit4 它的优点是速度快，支持代码覆盖率如 jacoco 等代码质量的检测工具。缺点就是无法单独对 Android UI，一些类进行操作，与原生 Java 有一些差异。

2、Mockito：

可以使用 mock()方法模拟各种各样的对象，以替代真正的对象做出希望的响应。除此之外，它还有很多验证方法调用的方式如 Mockit.when(调用方法).thenReturn(验证的返回值)、verfiy(模拟对象).验证方法等等。

这里有一点要补充下：简单的测试会使整体的代码更简洁，更可读、更可维护。如果你不能把测试写的很简单，那么请在测试时重构你的代码。

最后，对于 Mockito 来说，它的优点是有各种各样的方式去验证"模仿对象"的互动或验证发生的某些行为。而它的缺点就是不支持 mock 匿名类、final 类、static 方法 private 方法。

3、PowerMockito：

因此，为了解决 Mockito 的缺陷，PoweMockito 出现了，它扩展了 Mockito，支持 mock 匿名类、final 类、static 方法、private 方法。只要使用它提供的 api 如 PowerMockito.mockStatic()去 mock 含静态方法或字段的类，PowerMockito.suppress(PowerMockito.method(类.class， 方法名)即可。

4、Robolectric

前面 3 种我们说的都是 Java 相关的单元测试方法，如果想在 Java 单元测试里面进行 Android 单元测试，还得使用 Robolectric，它提供了一套能运行在 JVM 的 Android 代码。它提供了一系列类似 ShadowToast.getLatestToast()、ShadowApplication.getInstance()这种方式来获取 Android 平台对应的对象。可以看到它的优点就是支持大部分 Android 平台依赖类的底层引用与模拟。缺点就是在异步测试的情况下有些问题，这是可以结合 Mockito 来将异步转为同步即可解决。

最后，自动化 UI 测试项目中我使用的是 Expresso，它提供了一系列类似 onView().check().perform()的方式来实现点击、滑动、检测页面显示等自动化的 UI 测试效果，这里在我的 WanAndroid 项目下的 BasePageTest 基类里面封装了一系列通用的方法，有兴趣可以去看看。

33、Jenkins 持续集成。

34、工作中有没有用过或者写过什么工具？脚本，插件等等；比如：多人协同开发可能对一些相同资源都各自放了一份，有没有方法自动检测这种重复之类的。

35、如何绕过 9.0 限制？

如何限制？

- 1、阻止 java 反射和 JNI。
- 2、当获取方法或 Field 时进行检测。
- 3、怎么检测？

区分出是系统调用还是开发者调用：

根据堆栈，回溯 Class，查看 ClassLoader 是否是 BootStrapClassLoader。

区分后，再区分是否是 hidden api：

Method，Field 都有 access_flag，有一些备用字段，hidden 信息存储其中。

如何绕过？

1、不用反射：

利用一个 fakelib，例如写一个 android.app.ActivityThread#currentActivityThread 空实现，直接调用；

2、伪装系统调用：

jni 修改一个 class 的 classloder 为 BootStrapClassLoader，麻烦。

利用系统方法去反射：

利用原反射，即：getDeclaredMethod 这个方法是系统的方法，通过

getDeclaredmethod 反射去执行 hidden api。

3、修改 Method，Field 中存储 hidden 信息的字段：

利用 jni 去修改。

36、对文件描述符怎么理解？

37、如何实现进程安全写文件？

第五章 其他扩展面试题

一、Kotlin （★★）

1、Kotlin 特性，和 Java 相比有什么不同的地方？

- 能直接与 Java 相互调用，能与 Java 工程共存
- 大大减少样板代码
- 可以将 Kotlin 代码编译为无需虚拟机就可运行的原生二进制文件
- 支持协程
- 支持高阶函数
- 语言层面解决空指针问题
- 对字符串格式化的处理（$变量名）
- 更像 Python 的语法
- 对λ表达式支持更好

https://mp.weixin.qq.com/s/FqXLNz5p9M-5vcMUkxJyFQ

2、Kotlin 为什么能和 Java 混编？

3、什么是协程？

二、大前端 （★★）

1、Hybrid 通信原理是什么，有做研究吗？

2、JS 的交互理解吗？平时工作用的多吗，项目中是怎么与 Web 交互的？

Android 通过 WebView 调用 JS 代码：

1、通过 WebView 的 loadUrl()：

设置与 Js 交互的权限：

webSettings.setJavaScriptEnabled(true)

设置允许 JS 弹窗：

webSettings.setJavaScriptCanOpenWindowsAutomatically(true)

载入 JS 代码：

mWebView.loadUrl("file:///android_asset/javascript.html")

webview 只是载体，内容的渲染需要使用 webviewChromClient 类去实现，通过设置 WebChromeClient 对象处理 JavaScript 的对话框。

特别注意：

JS 代码调用一定要在 onPageFinished（） 回调之后才能调用，否则不会调用。

2、通过 WebView 的 evaluateJavascript()：

- 该方法比第一种方法效率更高、使用更简洁，因为该方法的执行不会使页面刷新，而第一种方法（loadUrl ）的执行则会。
- Android 4.4 后才可使用。

只需要将第一种方法的 loadUrl()换成 evaluateJavascript()即可，通过 onReceiveValue()回调接收返回值。

建议：两种方法混合使用，即 Android 4.4 以下使用方法 1，Android 4.4 以上方法 2。

JS 通过 WebView 调用 Android 代码：

1、通过 WebView 的 addJavascriptInterface（）进行对象映射：

-定义一个与 JS 对象映射关系的 Android 类：AndroidtoJs：

- 定义 JS 需要调用的方法，被 JS 调用的方法必须加入@JavascriptInterface 注解。
- 通过 addJavascriptInterface()将 Java 对象映射到 JS 对象。

优点：使用简单，仅将 Android 对象和 JS 对象映射即可。

缺点：addJavascriptInterface 接口引起远程代码执行漏洞，漏洞产生原因是：

当 JS 拿到 Android 这个对象后，就可以调用这个 Android 对象中所有的方法，包括系统类（java.lang.Runtime 类），从而进行任意代码执行。

2、通过 WebViewClient 的方法 shouldOverrideUrlLoading ()回调拦截 url：

Android 通过 WebViewClient 的回调方法 shouldOverrideUrlLoading () 拦截 url

解析该 url 的协议。

如果检测到是预先约定好的协议，就调用相应方法。

根据协议的参数，判断是否是所需要的 url。 一般根据 scheme（协议格式） & authority（协议名）判断（前两个参数）。

优点：不存在方式 1 的漏洞；

缺点：JS 获取 Android 方法的返回值复杂,如果 JS 想要得到 Android 方法的返回值，只能通过 WebView 的 loadUrl （）去执行 JS 方法把返回值传递回去。

3、通过 WebChromeClient 的 onJsAlert()、onJsConfirm()、onJsPrompt（）方法回调拦截 JS 对话框 alert()、confirm()、prompt（） 消息：

原理：

Android 通过 WebChromeClient 的 onJsAlert()、onJsConfirm()、onJsPrompt（）方法回调分别拦截 JS 对话框 （警告框、确认框、输入框），得到他们的消息内容，然后解析即可。

常用的拦截是：拦截 JS 的输入框（即 prompt（）方法），因为只有 prompt（）可以返回任意类型的值，操作最全面方便、更加灵活；而 alert（）对话框没有返回值；confirm（）对话框只能返回两种状态（确定 / 取消）两个值。

Android：你要的 WebView 与 JS 交互方式 都在这里了

3、react native 有多少了解？讲一下原理。

4、weex 了解吗？如何自己实现类似技术？

5、flutter 了解吗？内部是如何实现跨平台的？如何实现多 Native 页面接入？如何实现对现有工程的 flutter 迁移？

6、Dart 语言有研究过吗？

7、快应用了解吗？跟其她方式相比有什么优缺点？

8、说说你用过的混合开发技术有哪些？各有什么优缺点？

三、脚本语言 （★★）

1、脚本语言会吗?

2、Python 会吗?

Python 基础

人工智能了解

3、Gradle 了解多少? groovy 语法会吗?

第六章非技术面试题

2、你还要什么了解和要问的吗?

你在公司的一天是如何度过的?

能否给我简单介绍下贵公司业务与战略的未来发展?

贵公司最让你自豪的企业文化是什么?(适合大公司)

团队、公司现在面临的最大挑战是什么?

对于未来加入这个团队,你对我的期望是什么?

您觉得我哪方面知识需要深入学习或者我的不足在那些方面,今后我该注意什么*?

你还可以问下项目团队多少人,主要以什么方向为主,一年内的目标怎样,团队气氛怎样,等内容着手。

一、高频题集 (★ ★ ★)

1、你觉得安卓开发最关键的技术在哪里?

技术是没有止境的,所以肯定会不断有演进和难点。

一. 底层和框架如何更好地设计及优化以适应业务的高速增长。说起来很简单，低耦合高扩展，做起来是需要长期经验积累。

二. 我抛几个细节难点：

- 插件化如何使插件的 Manifest 生效
- H5 容器如何更好地优化和兼容
- App 端优化，这是个没止境的话题，网络、图片、动画、内存、电量等等随着优化的加深，你会发现不能局限在客户端，服务端也需要深入。
- SPDY 的优点并入 HTTP 2.0 你们有在测试或用吗？
- Fresco 出来前你是不是觉得图片缓存已经到头了？
- Android App 为什么整体流畅性总是被诟病？.....

三. 如果你觉得没有难点或者难点在兼容、UI 之类问题上，那么可能两个原因：

- 公司业务发展过慢，对技术的需求不够迫切
- 个人长时间在业务开发上，这个对于走技术路线的人来说挺麻烦的，不主动去接触学习的话，n 年以后也还是这个样子为了更好的个人成长，这两点都是需要注意和解决的问题。

3、研究比较深入的领域有哪些？

4、自己最擅长的技术点，最感兴趣的技术领域和技术？

5、项目中用了哪些开源库，如何避免因为引入开源库而导致的安全性和稳定性问题？

6、说下你都看过那些技术书籍，你是如何自学的。你觉得自己的优势与弱点是什么。

7、说下项目中遇到的棘手问题，包括技术，交际和沟通。

8、说下你近几年的规划？

9、对加班怎么看（不要太浮夸，现实一点哦）？

10、介绍你做过的哪些项目。

11、你并非毕业于名牌院校？

12、为什么要离职？

13、当你的开发任务很紧张，你怎么去做代码优化的？

二、次高频题集 （★★）

1、对业内信息的关注渠道有哪些？

2、最近都读哪些书？

3、给你一个项目，你怎么看待他的市场和技术的关系？

4、你以往的项目中，以你现在的眼光去评价项目的利弊？

5、对于非立项（KPI）项目，怎么推进？

6、都使用过哪些自定义控件？

7、除了简历上的工作经历，您还会去关注哪些领域？

8、评价下自己，评价下自己的技术水平，个人代码量如何？

9、你朋友对你的评价？

10、自己的优点和缺点是什么？并举例说明？

11、你觉得你个性上最大的优点是什么？

12、说说你最大的缺点？

13、最能概括你自己的三个词是什么？

14、说说你的家庭？

15、除了本公司外，还应聘了哪些公司？（类似问题：当前的 offer 状况）

16、通过哪些渠道了解的招聘信息？

17、你的业余爱好是什么？

18、你做过的哪件事最令自己感到骄傲？

19、谈谈你对跳槽的看法？

20、怎样看待学历和能力？

21、您跟您的主管或直接上司有没有针对以上离职原因的这些问题沟通过？如果没有请说明原因。如果有请说一下过程和结果？

22、您觉得你关注的这些领域跟您目前从事的职业有哪些利弊关系？如果有请说明利弊关系？

23、您在选择工作中更看重的是什么？（可能是成长空间、培训机会、发挥平台、薪酬等答案）

24、您可不可以说说您在薪酬方面的心里预期？

25、有人说挣未来比挣钱更为重要，您怎么理解？

26、假设，某一天，在工作办公室走廊，您和一位同事正在抱怨上级陈某平时做事缺乏公平性，恰巧被陈某听到，您会怎么办？

27、怎么样处理工作和生活的关系？怎么处理在工作中遇到困难？请举例说明

28、在您的现实生活中，您最不喜欢和什么样的人共事？为什么？举例说明。

29、在您认识的人中，有没有人不喜欢您？为什么不喜欢您？请举例说明。

30、当老板/上司/同事/客户误会你，你会怎么办？

31、当你发现其他部门的工作疏漏已经影响到您的工作绩效时，您怎么办？

32、您希望在什么样的领导下工作？

33、我们工作与生活历程并不是一帆风顺的，谈谈您的工作或生活中出现的挫折或低潮期，您如何克服？

34、假如您的上司是一个非常严厉、领导手腕强硬，时常给您巨大压力的人，您觉得这种领导方式对您有何利、弊？

35、您的领导给您布置了一项您以前从未触及过的任务，您打算如何去完成它？（如果有类似的经历说说完成的经历。）

36、谈谈您以往职业生涯中最有压力的一、两件事，并说说是如何克服的。

37、谈谈您以往职业生涯中令您有成就感的一、两件事，并说说它给您的启示。

38、请您举一个例子，说明在完成一项重要任务时，您是怎样和他人进行有效合作的。

39、当你要牺牲自己的某些方面与他人共事时，你会怎么办？

40、有时团队成员不能有效共事，当遇到这种问题时你是怎么处理的？你又是如何改善这类情

况的？

41、我们有时不得不与自己不喜欢的人在一个团队工作，如果遇到这样的情况你会怎么办？

42、您对委任的任务完成不了时如何处理？

43、说说您对下属布置的任务在时间方面是如何要求的？

44、说说您在完成上司布置的任务时，在时间方面是如何要求自己的？

45、您以往在领导岗位中，一个月内分别有哪些主要的工作任务？

46、当您发现您的部属目前士气较低沉，您一般从哪些方面去调动？

47、说说您在以往领导岗位中出现管理失控的事例及事后的原因分析。您的部属在一个专业的问题上跟您发生争议，您如何对待这种事件？

48、你对某某某互联网发生事情的看法？（直播答题等等）

49、怎么看待前端和后端？