Android Study Plan

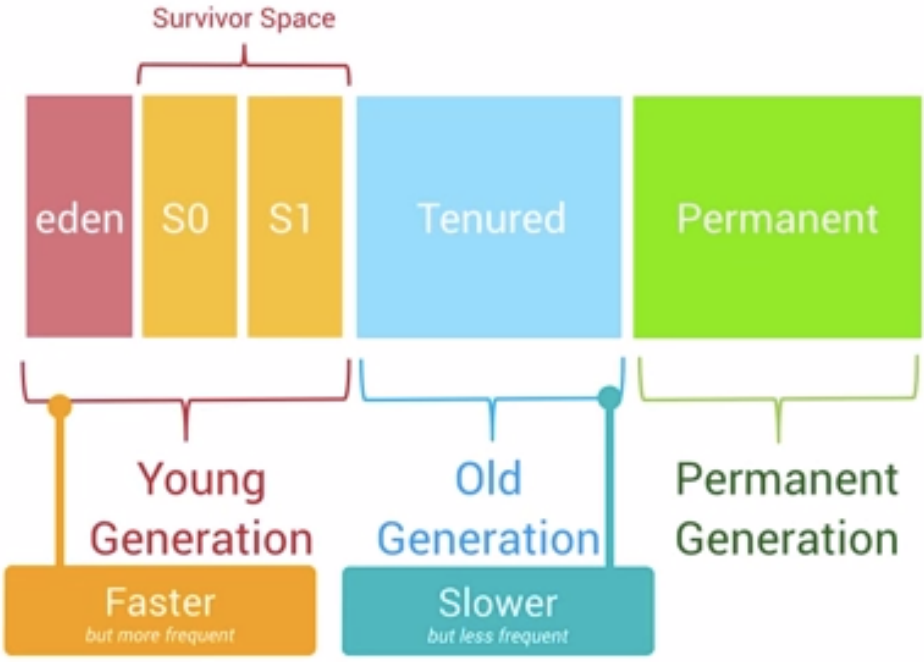
deepwaterooo

2021 年 12 月 19 日

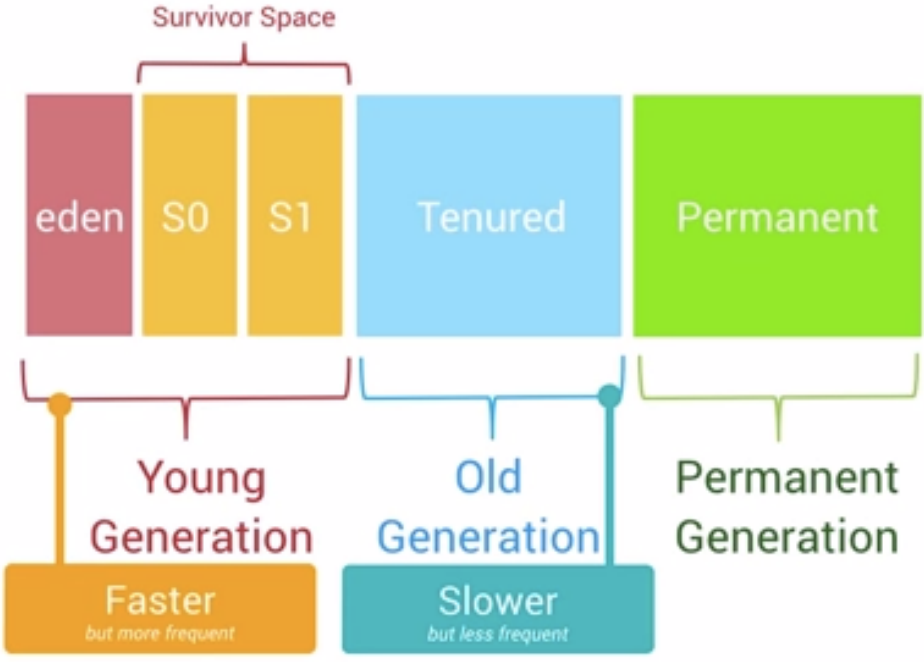
目录

1 性能优化

- https://segmentfault.com/a/1190000037449196 总结得很全面也很透彻


Survivor Space
eden
S0
S1
Tenured
Permanent
Young Generation
Old Generation
Permanent Generation
Faster
but more frequent
Slower
but less frequent

1.1 内存优化


Survivor Space
eden
S0
S1
Tenured
Permanent
Young Generation
Old Generation
Permanent Generation
Faster
but more frequent
Slower
but less frequent

- 这就是 Android 开发童鞋需要了解的 Generational Heap Memory 模型，这里我们只关心当对象在 Young Generation 中存活了一段时间之后，如果没被干掉，那么会被移动到 Old Generation 中，同理，最后会移动到 Permanent Generation 中。

1.1.1 珍惜 **Service**，尽量使得 **Service** 在使用的时候才处于运行状态。尽量使用 **IntentService**

- IntentService 在内部其实是通过线程以及 Handler 实现的，当有新的 Intent 到来的时候，会创建线程并且处理这个 Intent，处理完毕以后就自动销毁自身。因此使用 IntentService 能够节省系统资源。

1.1.2 内存紧张的时候释放资源（例如 **UI** 隐藏的时候释放资源等）。复写 **Activity** 的回调方法。

```
@Override
public void onLowMemory() {
    super.onLowMemory();
}
@Override
public void onTrimMemory(int level) {
    super.onTrimMemory(level);
    switch (level) {
    case TRIM_MEMORY_COMPLETE:
        break;
    case 其他:
    }
}
```

1.1.3 通过 **Manifest** 中对 **Application** 配置更大的内存，但是一般不推荐

```
android:largeHeap="true"
```

1.1.4 避免 **Bitmap** 的浪费，应该尽量去适配屏幕设备。尽量使用成熟的图片加载框架，**Picasso**，**Fresco**，**Glide** 等。

1.1.5 使用优化的容器，**SparseArray** 等

- Android 为什么推荐使用 SparseArray 来替代 HashMap？
- SparseArray 有两个优点：
  - 1. 避免了自动装箱（auto-boxing）
  - 2. 数据结构不会依赖于外部对象映射。我们知道 HashMap 采用一种所谓的“Hash 算法”来决定每个元素的存储位置，存放的都是数组元素的引用，通过每个对象的 hash 值来映射对象。而 SparseArray 则是用数组数据结构来保存映射，然后通过折半查找来找到对象。但其实一般来说，SparseArray 执行效率比 HashMap 要慢一点，因为查找需要折半查找，而添加删除则需要在数组中执行，而 HashMap 都是通过外部映射。但相对来说影响不大，最主要是 SparseArray 不需要开辟内存空间来额外存储外部映射，从而节省内存。

1.1.6 其他建议：尽量少用枚举变量，尽量少用抽象，尽量少增加类，避免使用依赖注入框架，谨慎使用 **library**，使用代码混淆，时当场合考虑使用多进程等。

1.1.7 避免内存泄漏（本来应该被回收的对象没有被回收）

- 一旦 APP 的内存短时间内快速增长或者 GC 非常频繁的时候，就应该考虑是否是内存泄漏导致的。

1. 什么情况会导致内存泄漏

- 1. 资源释放问题：程序代码的问题，长期保持某些资源，如 Context、Cursor、IO 流的引用，资源得不到释放造成内存泄露。
  - 资源对象没有关闭引起的内存泄漏。在这些资源不使用的时候，记得调用相应的类似 close ()、destroy ()、recycler ()、release () 等方法释放。
  - 注册/反注册未成对使用引起的内存泄漏。注册广播接受器、EventBus 等，记得解绑。
- 2. 对象内存过大问题：保存了多个耗用内存过大的对象（如 Bitmap、XML 文件），造成内存超出限制。
- 3.static 关键字的使用问题：static 是 Java 中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用 static 修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（Context 的情况最多），这时就要谨慎对待了。
  - 解决方案
    - 应该尽量避免 static 成员变量引用资源耗费过多的实例，比如 Context。
    - Context 尽量使用 ApplicationContext，因为 Application 的 Context 的生命周期比较长，引用它不会出现内存泄露的问题。
    - 使用 WeakReference 代替强引用。比如可以使用 WeakReference<Context> mContextRef
- 4. 线程导致内存溢出：线程产生内存泄露的主要原因在于线程生命周期的不可控。例如 Activity 中的 Thread 在 run 了，但是 Activity 由于某种原因重新创建了，但是 Thread 仍然会运行，因为 run 方法不结束的话 Thread 是不会销毁的。
  - Handler 引起的内存泄漏。解决：将 Handler 声明为静态内部类，就不会持有外部类 SecondActivity 的引用，其生命周期就和外部类无关，如果 Handler 里面需要 context 的话，可以通过弱引用方式引用外部类
  - 解决方案
    - 将线程的内部类，改为静态内部类（因为非静态内部类拥有外部类对象的强引用，而静态类则不拥有）。
    - 在线程内部采用弱引用保存 Context 引用。

2. 内存泄漏的分析工具与方法

- 使用 Android Studio 提供的 Android Monitors 中 Memory 工具查看内存的使用以及没使用的情况。
- 使用 DDMS 提供的 Heap 工具查看内存使用情况，也可以手动触发 GC。
- 使用性能分析的依赖库，例如 Square 的 LeakCanary，这个库会在内存泄漏的前后通过 Notification 通知你。

(a) LeakCanary 原理:

- 它的基本工作原理如下：
  - RefWatcher.watch() 创建一个 KeyedWeakReference 到要被监控的对象。
  - 然后在后台线程检查引用是否被清除，如果没有，调用 GC。
  - 如果引用还是未被清除，把 heap 内存 dump 到 APP 对应的文件系统中的一个.hprof 文件中。
  - 在另外一个进程中的 HeapAnalyzerService 有一个 HeapAnalyzer 使用 HAHA 解析这个文件。
  - 得益于唯一的 reference key, HeapAnalyzer 找到 KeyedWeakReference，定位内存泄漏。

- HeapAnalyzer 计算到 GC roots 的最短强引用路径，并确定是否是泄漏。如果是的话，建立导致泄漏的引用链。
- 引用链传递到 APP 进程中的 DisplayLeakService，并以通知的形式展示出来。

(b) 总的来说，LeakCanary 有如下几个明显优点：

- 针对 Android Activity 组件完全自动化的内存泄漏检查。
- 可定制一些行为（dump 文件和 leaktrace 对象的数量、自定义例外、分析结果的自定义处理等）。
- 集成到自己工程并使用的成本很低。
- 友好的界面展示和通知。

1.2 性能优化

- 防止过度绘制，通过打开手机的“显示过度绘制区域”即可查看过度绘制的情况。
- 最小化渲染时间，使用视图树查看节点，对节点进行性能分析。
- 通过 TraceView 进行数据的采集以及分析。在有大概定位的时候，使用 Android 官方提供的 Debug 类进行采集。*** 通过 DDMS 即可打开这个.trace 文件，分析函数的调用情况（包括在指定情况下执行时间，调用次数）

```
// 开启数据采集
Debug.startMethodTracing("test.trace");
// 关闭
Debug.stopMethodTracing();
```

1.3 OOM

- 避免 OOM 的一些常见方法：
  - 1. App 资源中尽量少用大图。使用 Bitmap 的时候要注意等比例缩小图片，并且注意 Bitmap 的回收。

```
BitmapFactory.Options options = new BitmapFactory.Option();
options.inSampleSize = 2;
//Options 只保存图片尺寸大小，不保存图片到内存
BitmapFactory.Options opts = new BitmapFactory.Options();
opts.inSampleSize = 2;
Bitmap bmp = null;
bmp = BitmapFactory.decodeResource(getResources(),
mImageIds[position],opts);
//回收
bmp.recycle();
```

- 2. 结合组件的生命周期，释放资源
- 3. IO 流，数据库查询的游标等应该在使用完之后及时关闭。
- 4. ListView 中应该使用 ViewHolder 模式缓存 ConverView
- 5. 页面切换的时候尽量去传递（复用）一些对象

1.4 ANR

- 不同的组件发生 ANR 的时间不一样，主线程（Activity、Service）是 5 秒，BroadCastReceiver 是 10 秒。
- ANR 一般有三种类型：
  - 1. KeyDispatchTimeout(5 seconds)
    - 主要类型按键或触摸事件在特定时间内无响应
  - 2. BroadcastTimeout(10 seconds)
    - BroadcastReceiver 在特定时间内无法处理完成
  - 3. ServiceTimeout(20 seconds)
    - 小概率类型 Service 在特定的时间内无法处理完成
- 解决方案：
  - 1. UI 线程只进行 UI 相关的操作。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、runonUITread、AsyncTask 等方式更新 UI。
  - 2. 无论如何都要确保用户界面操作的流畅度。如果耗时操作需要让用户等待，那么可以在界面上显示进度条。
  - 3. BroadCastReceiver 要进行复杂操作的的时候，可以在 onReceive() 方法中启动一个 Service 来处理。

1.5 内存抖动

- Gc 引起卡顿 +OOM，怎么优化
- Gson 反序列化导致产生大量对象
- 解决思考：对象池

1.6 网络请求优化

- 能够缓存起来的尽量去缓存起来，减轻服务器的压力。例如 APP 中首页的一些数据，又例如首页的图标、文案都是缓存起来的，而且这些数据通过网络来指定可以使 app 具有更大的灵活性。
- 不用域名，用 IP 直连，省去了 DNS 域名解析。
- 连接复用、请求合并、请求数据 Body 可以利用压缩算法 Gzip 来进行压缩，使用 JSON 代替 XML
- 网络请求的安全性: 这块了解的不多。我给你说说我的思路吧，利用哈希算法，比如 MD5，服务器给我们的数据可以通过时间戳和其他参数做个加密，得到一个 key，在客户端取出数据后根据数据和时间戳再去生成 key 与服务端给的做个对比。

1.7 导致性能问题的原因

- 1. 人为在 ui 线程中做了轻微的耗时操作，导致 ui 线程卡顿
- 2.layout 过于复杂，无法在 16ms 完成渲染
  - 先简单的把渲染大概分为"layout","measure""draw" 这么几个阶段，当然你不要以为实际情况也是如此，好，层级复杂，layout,measure 可能就用到了不该用的时间，自然而然，留给 draw 的时间就可能不够了，自然而然就悲剧了。
  - 那么以前给出的很多建议是，使用 RelativeLayout 替换 LinearLayout, 说是可以减少布局层次，然鹅，现在请不要在建议别人使用 RelativeLayout，因为 ConstraintLayout 才是一个更高性能的消灭布局层级的神器。ConstraintLayout 基于 Cassowary 算法，而 Cassowary 算法的优势是在于解决线性方程时有极高的效率，事实证明，线性方程组是非常适合用于定义用户界面元素的参数。由于人们对图形的敏感度非常高，所以 UI 的渲染速度显得非常重要。
- 3. 同一时间执行的动画过多，导致 CPU 或者 GPU 负载过重
  - 这里主要是因为动画一般会频繁变更 view 的属性，导致 displayList 失效，而需要重新创建一个新的 displayList，如果动画过多，这个开销可想而知，如果你想了解得更加详细，推荐看这篇咯，知识点在第 5 节那里。
- 4.view 过度绘制的问题。
  - view 过度绘制的问题可以说是我们在写布局的时候遇到的一个最常见的问题之一，可以说写着写着一不留神就写出了一个过度绘制，通常发生在一个嵌套的 viewgroup 中，比如你给他设置了一个不必要的背景。这方面问题的排查不太难，我们可以通过手机设置里面的开发者选项，打开 Show GPU Overdraw 的选项，轻松发现这些问题，然后尽量往蓝色靠近。
- 5.gc 过多的问题，这里就不在赘述了，上面已经讲的非常直接了。
- 6、资源加载导致执行缓慢。
- 有些时候避免不要加载一些资源，这里有两种解决的办法，使用的场景也不相同。
  - a、预加载，即还没有来到路径之前，就提前加载好，诶，好像 x5 内核就是酱紫哦。
  - b、实在是要等到用到的时候加载，请给一个进度条，不要让用户干等着，也不知道什么时候结束而造成不好的用户体验。
- 7、工作线程优先级设置不对，导致和 ui 线程抢占 cpu 时间。

- 使用 Rxjava 的小伙伴要注意这点，设置任务的执行线程可能会对你的性能产生较大的影响，没有使用的小伙伴也不能太过大意。
- 8、静态变量。
  - 嘿嘿，大家一定有过在 application 中设置静态变量的经历，遥想当年，为了越过 Intent 只能传递 1M 以下数据的坑，我在 application 中设置了一个静态变量，用于两个 activity “传递（共享）数据”，然而，一步小心，数据中，有着前一个 activity 的尾巴，因此泄露了。不光是这样的例子，随便举几个：
    * a、你用静态集合保存过数据吧？
    * b、某某单例的 Manger，比如管理 AudioManger 遇到过吧？

1.8 如何解决这些问题

1.8.1 1、GPU 过度绘制，定位过度绘制区域

- 这里直接在开发者选项，打开 Show GPU Overdraw，就可以看到效果，轻松发现哪块需要优化，那么具体如何去优化
  - a、减少布局层级，上面有提到过，使用 ConstraintLayout 替换传统的布局方式。如果你对 ConstraintLayout 不了解，没有关系，这篇文章教你 15 分钟了解如何使用 ConstraintLayout。
  - b、检查是否有多余的背景色设置，我们通常会犯一些低级错误--对被覆盖的父 view 设置背景，多数情况下这些背景是没有必要的。

1.8.2 2、主线程耗时操作排查。

- a、开启 strictmode, 这样一来，主线程的耗时操作都将以告警的形式呈现到 logcat 当中。
- b、直接对怀疑的对象加 @DebugLog 注解，查看方法执行耗时。DebugLog 注解需要引入插件 hugo，这个是 Android 之神 JakeWharton 的早期作品，对于监控函数执行时间非常方便，直接在函数上加入注解就可以实现，但是有一个缺点，就是 JakeWharton 发布的最后一个版本没有支持 release 版本用空方法替代监控代码，因此，我这里发布了一个到公司的 maven 仓库，引用的方式和官网类似，只不过，地址是：

```
'com.tencent.tip:hugo-plugin:2.0.0-SNAPSHOT'。
```

1.8.3 3、对于 **measure，layout** 耗时过多的问题

- 一般这类问题是优于布局过于复杂的原因导致，现在因为有 ConstraintLayout，所以，强烈建议使用 ConstraintLayout 减少布局层级，问题一般得以解决，如果发现还存在性能问题，可以使用 traceView 观察方法耗时，来定位下具体原因。

1.8.4 4、**leakcany**

- 这个是内存泄露监测的银弹，大家应该都使用过，需要提醒一下的是，要注意

```
dependencies {
    debugImplementation 'com.squareup.leakcanary:leakcanary-android:1.5.4'
    releaseImplementation 'com.squareup.leakcanary:leakcanary-android-no-op:1.5.4'
}
```

- 引入方式，releaseImplementation 保证在发布包中移除监控代码，否则，他自生不停的 catch 内存快照，本身也影响性能。

1.8.5 5、**onDraw** 里面写代码需要注意

- onDraw 优于大概每 16ms 都会被执行一次，因此本身就相当于一个 forloop，如果你在里面 new 对象的话，不知不觉中就满足了短时间内大量对象创建并释放，于是频繁 GC 就发生了，嗯，内存抖动，于是，卡了。因此，正确的做法是将对象放在外面 new 出来。

1.8.6 6、**json** 反序列化问题

- json 反序列化是指将 json 字符串转变为对象，这里如果数据量比较多，特别是有相当多的 string 的时候，解析起来不仅耗时，而且还很吃内存。解决的方式是：
  - a、精简字段，与后台协商，相关接口剔除不必要的字段。保证最小可用原则。
  - b、使用流解析，之前我考虑过 json 解析优化，在 Stack Overflow 上搜索到这个。于是了解到 Gson.fromJson 是可以这样玩的，可以提升 25% 的解析效率。

```
public List<Messge> readJsonStream(InputStream in) throws IOException {
    JsonReader reader = new JsonReader(enw InputStreamReader(in, "UTF-8"));
    List<Message> msgs = new ArrayList<>();
    reader.beginArray();
    while (reader.hasNext()) {
        Message msgCur = gson.fromJson(reader, Message.class);
        msgs.add(msgCur);
    }
    reader.endArray();
    reader.close();
    return msgs;
}
```

1.8.7 7、**viewStub & merge** 的使用

- 这里 merge 和 viewStub 想必是大家非常了解的两个布局组件了，对于只有在某些条件下才展示出来的组件，建议使用 viewStub 包裹起来，同样的道理，include 某布局如果其根布局和引入他的父布局一致，建议使用 merge 包裹起来，如果你担心 preview 效果问题，这里完全没有必要，因为你可以

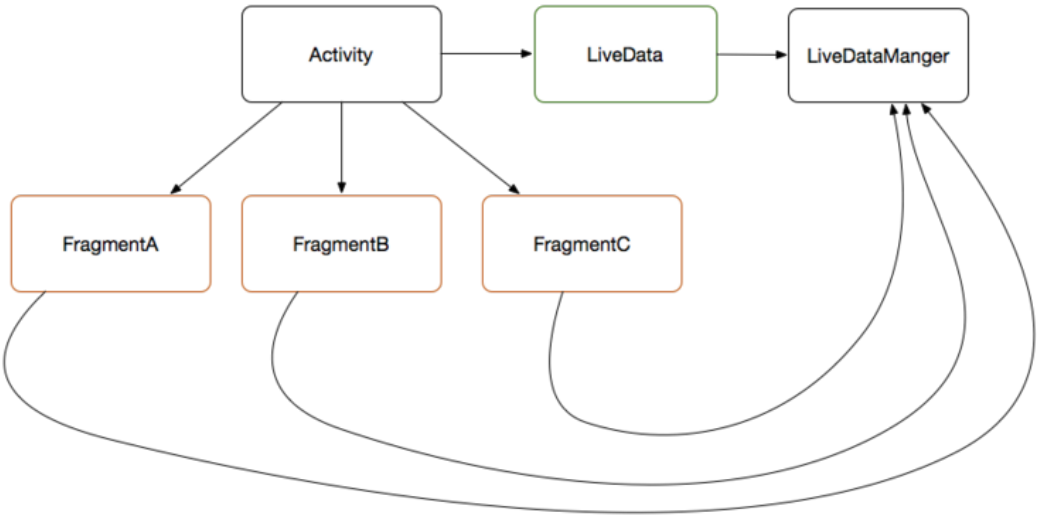
- tools:showIn="" 属性，这样就可以正常展示 preview 了。

1.8.8 8、加载优化

- 这里并没有过多的技术点在里面，无非就是将耗时的操作封装到异步中去了，但是，有一点不得不提的是，要注意多进程的问题，如果你的应用是多进程，你应该认识到你的 application 的 oncreate 方法会被执行多次，你一定不希望资源加载多次吧，于是你只在主进程加载，如是有些坑就出现了，有可能其他进程需要那某份资源，然后他这个进程缺没有加载相应的资源，然后就嗝屁了。
- 如果你的应用是多进程，你应该认识到你的 application 的 oncreate 方法会被执行多次。多进程应用，概念、应用设计等还需要好好熟悉一下

1.8.9 9、刷新优化。

- 这点在我之前的文章中有提到过，这里举两个例子吧。
  - a、对于列表的中的 item 的操作，比如对 item 点赞，此时不应该让整个列表刷新，而是应该只刷新这个 item，相比对于熟练使用 recyclerView 的你，应该明白如何操作了, 不懂请看这里，你将会明白什么叫做 recyclerView 的局部刷新
  - b、对于较为复杂的页面，个人建议不要写在一个 activity 中，建议使用几个 fragment 进行组装，这样一来，module 的变更可以只刷新某一个具体的 fragment，而不用整个页面都走刷新逻辑。但是问题来了，fragment 之间如何共享数据呢？好，看我怎么操作。



- Activity 将数据这部分抽象成一个 LiveData，交个 LiveDataManger 数据进行管理，然后各个 Fragment 通过 Activity 的这个 context 从 LiveDataManger 中拿到 LiveData, 进行操作，通知 activity 数据变更等等。哈哈，你没有看错，这个确实和 Google 的那个 LiveData 有点像，当然，如果你想使用 Google 的那个，也自然没问题，只不过，这个是简化版的。项目的引入

```
'com.tencent.tip:simple_live_data:1.0.1-SNAPSHOT'
```

1.8.10 10、动画优化

- 这里主要是想说使用硬件加速来做优化，不过要注意，动画做完之后，关闭硬件加速，因为开启硬件加速本身就是一种消耗。下面有一幅图，第二幅对比第一幅是说开启硬件加速和没开启的时候做动画的效果对比，可以看到开启后的渲染速度明显快不少，开启硬件加速就一定万事

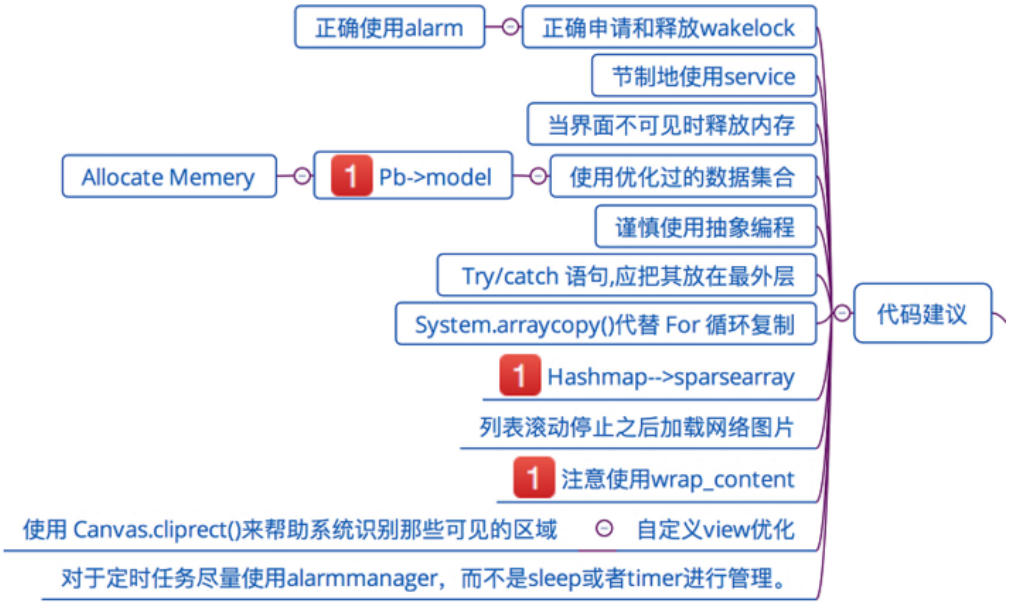
大吉么？第三幅图实际上就说明，如果你的这个 view 不断的失效的话，也会出现性能问题，第三图中可以看到蓝色的部曲线图有了一定的起色，这说明，displaylist 不断的失效并重现创建，如果你想了解的更加详细，可以查看这里

```
// Set the layer type to hardware
myView.setLayerType(View.LAYER_TYPE_HARDWARE, null);
// Setup the animation
ObjectAnimator animator = ObjectAnimator.ofFloat(myView,View.TRANSLATION_X, 150);
// Add a listener that does cleanup
animator.addListener(new AnimatorListenerAdapter() {
        @Override
        public void onAnimationEnd(Animator animation) {
            myView.setLayerType(View.LAYER_TYPE_NONE, null);
        }
    });
```

1.8.11 11、耗电优化

- 这里仅仅只是建议；
  - a、在定位精度要求不高的情况下，使用 wifi 或移动网络进行定位，没有必要开启 GPS 定位。
  - b、先验证网络的可用性，在发送网络请求，比如，当用户处于 2G 状态下，而此时的操作是查看一张大图，下载下来可能都 200 多 K 甚至更大，我们没必要去发送这个请求，让用户一直等待那个菊花吧。

1.9 关于一些代码的建议



- pb->model 这里的优化就不在赘述，前面有讲如何优化。
- 然后建议使用 SparseArray 代替 HashMap, 这里是 Google 建议的，因为 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 的自动装箱比如（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。

- 不到不得已,不要使用 $\text{wrap}_{content}$,, 推荐使用 match_{parent}, 或者固定尺寸,配合 gravity="center"。
- 因为在测量过程中,match_{parent} $EXACTLY$, 而 $\text{wrap}_{content}$ $ATMOST$, 这两者对比 AT_{MOST}

2 app 优化（项目中处理的一些难点）

- 主要分为启动优化，布局优化，打包优化等

2.1 启动优化

- 闪屏页优化，设置 theme 默认欢迎背景
- 懒加载第三方库，不要都放在 application 中初始化
- 如果项目中有 webview ，可以提前在 app 空闲时间加载 webview 的内核，如果多处使用可以创建缓存池，缓存 webview，
- 如果 android 5.0- 在 applicaton 的 attchbaseContext() 中加载 MultiDex.install 会更加耗时，可以采用子线程（子线程加载需要担心 ANR 和 ContentProvider 未加载报错的问题）或者单独开一个进程 B，进程 B 开启子线程运行 MultiDex.install ，让 applicaton 进入 while 循环等待 B 进程加载结果。
- MultiDex 优化，apk 打包分为 android 5.0 + 使用 ART 虚拟机不用担心

2.2 布局 UI 优化

- 看过布局绘制源码流程后，可以知道 setContextView 中在 ViewRootImpl 中使用 pull 的方法（这里可以扩展 xml 读取方式 SAX ：逐行解析、dom：将整个文件加载到内存然后解析，不推荐、pull：类似于 SAX 进行了 android 平台的优化，更加轻量级方便）迭代读取 xml 标签，然后对 view 进行 measure，layout 和 draw 的时候都存在耗时。通常优化方式有：
  - 减少 UI 层级、使用 merge、Viewstub 标签优化重复的布局
  - 优化 layout ，尽量多使用 ConstraintLayout，因为 relalayout 和 linearlayout 比重的情况下都存在多次测量
  - recyclerView 缓存 (可扩展说明 rv 的缓存原理)
  - 比较极端的将 measure 和 layout 放在子线程，在主线程进行 draw。或者子线程中加载 view 进行 IO 读取 xml, 通过 Handler 回调主线程加载 view(比如 android 原生类 AsyncLayoutInflate)
  - 将 xml 直接通过第三方工具（原理 APT 注解翻译 xml）直接将 xml 转为 java 代码

3 数据交互

3.1 Android 中数据常见存储方式

- 文件（包括 XML、SharePreference 等）
- 数据库
- Content Provider
- 保存在网络

3.2 进程间通信

3.2.1 操作系统进程间通信的方法，**android** 中有哪些？

- Windows：剪贴板、管道、邮槽等
- Linux：命名管道、共享内存、信号量

3.2.2 **Android** 中的进程通信方式并不是完全继承于 **Linux**：

- Android 跨进程通信，像 intent，contentProvider, 广播，service 都可以跨进程通信。
  - intent：这种跨进程方式并不是访问内存的形式，它需要传递一个 uri, 比如说打电话。
  - contentProvider：这种形式，是使用内存映射、数据共享的形式进行数据共享。
  - service：远程服务，aidl
- Intent 、Bundle ：要求传递数据能被序列化，实现 Parcelable、Serializable ，适用于四大组件通信。
- 文件共享：适用于交换简单的数据实时性不高的场景。
- AIDL：AIDL 接口实质上是系统提供给我们可以方便实现 BInder 的工具
- Messenger：基于 AIDL 实现，服务端串行处理，主要用于传递消息，适用于低并发一对多通信
- ContentProvider：基于 Binder 实现，适用于一对多进程间数据共享（通讯录短信等）
- Socket：TCP、UDP，适用于网络数据交换

3.2.3 **AIDL** 理解

- 此处延伸：简述 Binder
- AIDL: 每一个进程都有自己的 Dalvik VM 实例，都有自己的一块独立的内存，都在自己的内存上存储自己的数据，执行着自己的操作，都在自己的那片狭小的空间里过完自己的一生。而 aidl 就类似与两个进程之间的桥梁，使得两个进程之间可以进行数据的传输，跨进程通信有多种选择，比如 BroadcastReceiver , Messenger 等，但是 BroadcastReceiver 占用的系统资源比较多，如果是频繁的跨进程通信的话显然是不可取的；Messenger 进行跨进程通信时请求队列是同步进行的，无法并发执行。
- Binde 机制简单理解:
- 在 Android 系统的 Binder 机制中，是有 Client,Service,ServiceManager,Binder 驱动程序组成的，其中 Client，service，Service Manager 运行在用户空间，Binder 驱动程序是运行在内核空间的。而 Binder 就是把这 4 种组件粘合在一块的粘合剂，其中核心的组件就是 Binder 驱动程序，Service Manager 提供辅助管理的功能，而 Client 和 Service 正是在 Binder 驱动程序和 Service Manager 提供的基础设施上实现 C/S 之间的通信。其中 Binder 驱动程序提供设备文件/dev/binder 与用户控件进行交互，
- Client、Service，Service Manager 通过 open 和 ioctl 文件操作相应的方法与 Binder 驱动程序进行通信。而 Client 和 Service 之间的进程间通信是通过 Binder 驱动程序间接实现的。而 Binder Manager 是一个守护进程，用来管理 Service，并向 Client 提供查询 Service 接口的能力。

3.2.4 多线程间通信和多进程之间通信有什么不同，分别怎么实现?

1. 1、进程间的通信方式

   - 管道 (pipe): 管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
   - 有名管道 (namedpipe) : 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
   - 信号量 (semophore) : 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，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。
   - 消息队列 (messagequeue) : 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
   - 信号 (sinal) : 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
   - 共享内存 (shared memory) : 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。
   - 套接字 (socket) : 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

2. 2 . 线程间的通信方式

   - 锁机制: 包括互斥锁、条件变量、读写锁
     - 互斥锁提供了以排他方式防止数据结构被并发修改的方法。
     - 读写锁允许多个线程同时读共享数据，而对写操作是互斥的。
     - 条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。
   - 信号量机制 (Semaphore): 包括无名线程信号量和命名线程信号量
   - 信号机制 (Signal): 类似进程间的信号处理
   - 线程间的通信目的主要是用于线程同步，所以线程没有像进程通信中的用于数据交换的通信机制。

4 常见的网络框架

- HttpURLConnection: 在 Android 2.2 版本之前，HttpClient 拥有较少的 bug，因此使用它是 *** 的选择。而在 Android 2.3 版本及以后，HttpURLConnection 则是 *** 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我们也会将更多的时间放在优化 HttpURLConnection 上面。特点：比较轻便，灵活，易于扩展，在 3.0 后以及 4.0 中都进行了改善，如对 HTTPS 的支持，在 4.0 中，还增加了对缓存的支持。

- HttpClient：高效稳定，但是维护成本高昂，故 android 开发团队不愿意在维护该库而是转投更为轻便的

- okHttp：okhttp 是一个 Java 的 HTTP+SPDY 客户端开发包，同时也支持 Android。需要 Android 2.3 以上。特点：OKHttp 是 Android 版 Http 客户端。非常高效，支持 SPDY、连接池、GZIP 和 HTTP 缓存。默认情况下，OKHttp 会自动处理常见的网络问题，像二次连接、SSL 的握手问题。如果你的应用程序中集成了 OKHttp，Retrofit 默认会使用 OKHttp 处理其他网络层请求。从 Android4.4 开始 HttpURLConnection 的底层实现采用的是 okHttp。
- volley：早期使用 HttpClient，后来使用 HttpURLConnection，是谷歌 2013 年推出的网络请求框架，非常适合去进行数据量不大，但通信频繁的网络操作，而对于大数据量的网络操作，比如说下载文件等，Volley 的表现就会非常糟糕。
- xutils：缓存网络请求数据
- Retrofit：和 Volley 框架的请求方式很相似，底层网络请求采用 okhttp（效率高，android4.4 底层采用 okhttp），采用注解方式来指定请求方式和 url 地址，减少了代码量。

5 常用的图片加载框架以及特点、源码

- Picasso：PicassoSquare 的网络库一起能发挥 *** 作用，因为 Picasso 可以选择将网络请求的缓存部分交给了 okhttp 实现。
- Glide：模仿了 Picasso 的 API，而且在他的基础上加了很多的扩展 (比如 gif 等支持)，支持图片流，因此在做爱拍之类的视频应用用得比较多一些。
- Fresco：Fresco 中设计有一个叫做 image pipeline 的模块。它负责从网络，从本地文件系统，本地资源加载图片。为了 *** 限度节省空间和 CPU 时间，它含有 3 级缓存设计（2 级内存，1 级文件）。Fresco 中设计有一个叫做 Drawees 模块，方便地显示 loading 图，当图片不再显示在屏幕上时，及时地释放内存和空间占用。
- Fresco 是把图片缓存放在了 Ashmem（系统匿名内存共享区）
  - Heap-堆内存：Android 中每个 App 的 Java 堆内存大小都是被严格的限制的。每个对象都是使用 Java 的 new 在堆内存实例化，这是内存中相对安全的一块区域。内存有垃圾回收机制，所以当 App 不在使用内存的时候，系统就会自动把这块内存回收。不幸的是，内存进行垃圾回收的过程正是问题所在。当内存进行垃圾回收时，内存不仅仅进行了垃圾回收，还把 Android 应用完全终止了。这也是用户在使用 App 时最常见的卡顿或短暂假死的原因之一。
  - Ashmem：Android 在操作 Ashmem 堆时，会把该堆中存有数据的内存区域从 Ashmem 堆中抽取出来，而不是把它释放掉，这是一种弱内存释放模式；被抽取出来的这部分内存只有当系统真正需要更多的内存时（系统内存不够用）才会被释放。当 Android 把被抽取出来的这部分内存放回 Ashmem 堆，只要被抽取的内存空间没有被释放，之前的数据就会恢复到相应的位置。
- 不管发生什么，垃圾回收器都不会自动回收这些 Bitmap。当 Android 绘制系统在渲染这些图片，Android 的系统库就会把这些 Bitmap 从 Ashmem 堆中抽取出来，而当渲染结束后，这些 Bitmap 又会被放回到原来的位置。如果一个被抽取的图片需要再绘制一次，系统仅仅需要把它再解码一次，这个操作非常迅速。

6 View 相关

6.1 View 工作流程

- 通过 SetContentView()，调用到 PhoneWindow，后实例 DecorView，通过 LoadXmlResourceParser() 进行 IO 操作解析 xml 文件通过反射创建出 View，并将 View 绘制在 DecorView

上，这里的绘制则交给了 ViewRootImpl 来完成，通过 performTraversals() 触发绘制流程，performMeasure 方法获取 View 的尺寸，performLayout 方法获取 View 的位置，然后通过 performDraw 方法遍历 View 进行绘制。

6.2 事件分发

- 一个 MotionEvent 产生后，按 Activity -> Window -> DecorView（ViewGroup）-> View 顺序传递，View 传递过程就是事件分发，因为开发过程中存在事件冲突，所以需要熟悉流程：
  - dispatchTouchEvent：用于分发事件，只要接受到点击事件就会被调用，返回结果表示是否消耗了当前事件
  - onInterceptTouchEvent：用于判断是否拦截事件（只有 ViewGroup 中存在），当 ViewGroup 确定要拦截事件后，该事件序列都不会再触发调用此 ViewGroup 的 onIntercept
  - onTouchEvent：用于处理事件，返回结果表示是否处理了当前事件，未处理则传递给父容器处理。（事件顺序是：OnTouchListener -> OnTouchEvent -> OnClick）

6.3 自定义 **View!!**

准备自定义 View 方面的面试最简单的方法：就是自己动手实现几个 View（由简单到复杂）；分析一些热门 App 中的自定义 View 的效果是怎么实现的；阿里面试官：自定义 View 跟绘制流程相关知识点？（标准参考解答，值得收藏）

- https://www.cnblogs.com/Android-Alvin/p/12297933.html

7 **Retrofit** 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

8 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。

9 第三方库源码总结

9.1 LeakCanary 原理

参考博客: - https://www.jianshu.com/go-wild?ac=2&url=https%3A%2F%2Fwww.cnblogs.com%2Fjymblog%2Fp%2F11656221.html

- 通过 registerActivityLifecycleCallbacks 监听 Activity 或者 Fragment 销毁时候的生命周期（如果不想那个对象被监控则通过 AndroidExcludedRefs 枚举，避免被检测）

```
public void watch(Object watchedReference, String referenceName) {
   if (this == DISABLED) {
     return;
   }
   checkNotNull(watchedReference, "watchedReference");
   checkNotNull(referenceName, "referenceName");
   final long watchStartNanoTime = System.nanoTime();
   String key = UUID.randomUUID().toString();
   retainedKeys.add(key);
   final KeyedWeakReference reference =
       new KeyedWeakReference(watchedReference, key, referenceName, queue);
   ensureGoneAsync(watchStartNanoTime, reference);
}
```

- 然后通过弱引用和引用队列监控对象是否被回收（弱引用和引用队列 ReferenceQueue 联合使用时，如果弱引用持有的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。即 KeyedWeakReference 持有的 Activity 对象如果被垃圾回收，该对象就会加入到引用队列 queue）

```
void waitForIdle(final Retryable retryable, final int failedAttempts) {
    // This needs to be called from the main thread.
    Looper.myQueue().addIdleHandler(new MessageQueue.IdleHandler() {
      @Override public boolean queueIdle() {
        postToBackgroundWithDelay(retryable, failedAttempts);
        return false;
      }
    });
  }
```

- IdleHandler，就是当主线程空闲的时候，如果设置了这个东西，就会执行它的 queueIdle() 方法，所以这个方法就是在 onDestory 以后，一旦主线程空闲了，就会执行一个延时五秒的子线程任务，任务：检测到未被回收则主动 gc ，然后继续监控，如果还是没有回收掉，就证明是内存泄漏了。通过抓取 dump 文件，在使用第三方 HAHA 库分析文件，获取到到达泄露点最近的线路，通过启动另一个进程的 DisplayLeakService 发送通知进行消息的展示。

9.2 OkHttp

- 同步和异步网络请求使用方法

```
// 同步 get 请求
OkHttpClient okHttpClient=new OkHttpClient();
final Request request=new Request.Builder().url("xxx").get().build();
final Call call = okHttpClient.newCall(request);
try {
    Response response = call.execute();
} catch (IOException e) {
}

//异步 get 请求
OkHttpClient okHttpClient=new OkHttpClient();
final Request request=new Request.Builder().url("xxx").get().build();
final Call call = okHttpClient.newCall(request);
call.enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }
    @Override
    public void onResponse(Call call, Response response) throws IOException {
    }
});
// 异步 post 请求
OkHttpClient okHttpClient1 = new OkHttpClient();
RequestBody requestBody = new FormBody.Builder()
        .add("xxx", "xxx").build();
Request request1 = new Request.Builder().url("xxx").post(requestBody).build();
okHttpClient1.newCall(request1).enqueue(new Callback() {
    @Override
    public void onFailure(Call call, IOException e) {
    }
    @Override
    public void onResponse(Call call, Response response) throws IOException {
    }
});
```

同步请求流程：通过 OkHttpClient new 生成 call 实例 Realcall Dispatcher.executed() 中通过添加 realcall 到 runningSyncCalls 队列中通过 getResponseWithInterceptorChain() 对 request 层层拦截，生成 Response 通过 Dispatcher.finished()，把 call 实例从队列中移除，返回最终的 response 异步请求流程：生成一个 AsyncCall(responseCallback) 实例 (实现了 Runnable) AsyncCall 通过调用 Dispatcher.enqueue()，并判断 maxRequests （最大请求数）maxRequestsPerHost(最大 host 请求数) 是否满足条件，如果满足就把 AsyncCall 添加到 runningAsyncCalls 中，并放入线程池中执行；如果条件不满足，就添加到等待就绪的异步队列，当那些满足的条件的执行时，在 Dispatcher.finifshed(this) 中的 promoteCalls(); 方法中对等待就绪的异步队列进行遍历，生成对应的 AsyncCall 实例，并添加到 runningAsyncCalls 中，最后放入到线程池中执行，一直到所有请求都结束。

9.2.1 责任链模式和拦截器

责任链源码跟进 execute() 进入到 getResponseWithInterceptorChain() 方法 Response getResponseWithInterceptorChain() throws IOException { //责任链模式 List<Interceptor> interceptors = new ArrayList<>(); interceptors.addAll(client.interceptors()); interceptors.add(retryAndFoll

interceptors.add(new BridgeInterceptor(client.cookieJar())); interceptors.add(new CacheInterceptor(client.internalCache())); interceptors.add(new ConnectInterceptor(client)); if (!forWebSocket) { interceptors.addAll(client.networkInterceptors()); } interceptors.add(new CallServerInterceptor(forWebSocket)); Interceptor.Chain chain = new RealInterceptorChain(interceptors, null, null, null, 0, originalRequest, this, eventListener, client.connectTimeoutMillis(), client.readTimeou client.writeTimeoutMillis()); return chain.proceed(originalRequest); } chain.proceed() 方法核心代码。每个拦截器 intercept() 方法中的 chain，都在上一个 chain 实例的 chain.proceed() 中被初始化，并传递了拦截器 List 与 index，调用 interceptor.intercept(next)，直接最后一个 chain 实例执行即停止。//递归循环下一个拦截器 RealInterceptorChain next = new RealInterceptorChain(interceptors, streamAllocation, httpCodec, connection, index + 1, request, call, eventListener, connectTimeout, readTimeout, writeTimeout); Interceptor interceptor = interceptors.get(index); Response response = interceptor.intercept(next); @Override public Response intercept(Chain chain) throws IOException { Request request = chain.request(); RealInterceptorChain realChain = (RealInterceptorChain) chain; Call call = realChain.call(); EventListener eventListener = realChain.eventListener(); StreamAllocation streamAllocation = new StreamAllocation(client.connection createAddress(request.url()), call, eventListener, callStackTrace); while (true) { . . . // 循环中再次调用了 chain 对象中的 proceed 方法，达到递归循环。response = realChain.proceed(request, streamAllocation, null, null); releaseConnection = false; . . . } } 拦截器 RetryAndFollowUpInterceptor ： 重连并跟踪拦截器。BridgeInterceptor : 将用户请求构建为网络请求（hander cooker content-type 等）并发起请求。CacheInterceptor ： 缓存拦截器负责从缓存中返回响应和把网络请求响应写入缓存。ConnectInterceptor ： 与服务端建立连接，并且获得通向服务端的输入和输出流对象。OkHttp 流程采用责任链方式的拦截器, 实现分成处理网络请求, 可更好的扩展自定义拦截器（采用 GZIP 压缩，支持 http 缓存）采用线程池 (thread pool) 和连接池 (Socket pool) 解决多并发问题, 同时连接池支持多路复用 (http2 才支持, 可以让一个 Socket 同时发送多个网络请求, 内部自动维持顺序. 相比 http 只能一个一个发送, 更能减少创建开销)) 底层采用 socket 和服务器进行连接. 采用 okio 实现高效的 io 流读写

10 ArrayMap 和 HashMap 的区别

HashMap 和 ArrayMap 各自的优势 1. 查找效率: HashMap 因为其根据 hashcode 的值直接算出 index, 所以其查找效率是随着数组长度增大而增加的。ArrayMap 使用的是二分法查找, 所以当数组长度每增加一倍时, 就需要多进行一次判断, 效率下降。所以对于 Map 数量比较大的情况下, 推荐使用 2. 扩容数量: HashMap 初始值 16 个长度, 每次扩容的时候, 直接申请双倍的数组空间。ArrayMap 每次扩容的时候, 如果 size 长度大于 8 时申请 size*1.5 个长度, 大于 4 小于 8 时申请 8 个, 小于 4 时申请 4 个。这样比较 ArrayMap 其实是申请了更少的内存空间, 但是扩容的频率会更高。因此, 如果当数据量比较大的时候, 还是使用 HashMap 更合适, 因为其扩容的次数要比 ArrayMap 少很多。3. 扩容效率: HashMap 每次扩容的时候时重新计算每个数组成员的位置, 然后放到新的位置。ArrayMap 则是直接使用 System.arraycopy。所以效率上肯定是 ArrayMap 更占优势。这里需要说明一下, 网上有一种传闻说因为 ArrayMap 使用 System.arraycopy 更省内存空间, 这一点我真的没有看出来。arraycopy 也是把老的数组的对象一个一个的赋给新的数组。当然效率上肯定 arraycopy 更高, 因为是直接调用的 c 层的代码。4. 内存耗费：以 ArrayMap 采用了一种独特的方式, 能够重复的利用因为数据扩容而遗留下来的数组空间, 方便下一个 ArrayMap 的使用。而 HashMap 没有这种设计。由于 ArrayMap 只缓存了长度是 4 和 8 的时候, 所以如果频繁的使用到 Map, 而且数据量都比较小的时候,ArrayMap 无疑是相当的节省内存的。5. 总结: 综上所述, 数据量比较小, 并且需要频繁的使用 Map 存储数据的时候, 推荐使用 ArrayMap。而数据量比较大的时候, 则推荐使用 HashMap。

10.1 ButterKnife

参考文章 butterKnife 使用的是 APT 技术也就是编译时注解，不同于运行时注解（在运行过程中通过反射动态地获取相关类，方法，参数等信息，效率低），编译时注解则是在代码编译过程中对注解进行处理（annotationProcessor 技术），通过注解获取相关类，方法，参数等信息，然后在项目中生成代码，运行时调用，其实和直接手写代码一样，没有性能问题，只有编辑时效率问题。ButterKnife 在 Bind 方法中获取到 DecorView，然后通过 Activity 和 DecorView 对象获取 $xx_{ViewBinding}$ Constructor。在编写完 demo 之后，需要先 build 一下项目，之后可以在 build/generated/source/apt/debug/包名/下面找到对应的 $xx_{ViewBinding\ \ bk}$ 帮我们做的事情，$xx_{ViewBinding}$.java @UiThread public ViewActivity$_{ViewBinding}$(ViewActivity target, View source) { this.target = target; target.view = Utils.findRequiredView(source, R.id.view, "field 'view'"); } Utils.java public static View findRequiredView(View source, @IdRes int id, String who) { View view = source.findViewById(id); if (view != null) { return view; } String name = getResourceEntryName(source, id); throw new IllegalStateException("Required view ...."} 通过上述上述代码可以看到注解也是帮我们完成了 findviewbyid 的工作。butterknife 实现流程扫描 Java 代码中所有的 ButterKnife 注解发现注解，ButterKnifeProcessor 会帮你生成一个 Java 类，名字 < 类名 >$$ViewBinding.java，这个新生成的类实现了 Unbinder 接口，类中的各个 view 声明和添加事件都添加到 Map 中，遍历每个注解对应通过 JavaPoet 生成的代码。

10.2 Rxjava 2

切换到子线程用的线程池，切换到主线程则用的 Handler。底层的切换原理还是基于 Handler 来的。思路 1：在子线程发送消息，却能够在主线程接收消息，主线程和子线程是怎么样切换的？子线程用 handler 发送消息，发送的消息被送到与主线程相关联的 MessageQueue，也是主线程相关联的 Looper 在循环消息,handler 所关联的是主线程的 Looper 和 MessageQueue, 所以最后消息的处理逻辑也是在主线程。只有发送消息是在子线程，其它都是在主线程，Handler 与哪个线程的 Looper 相关联，消息处理逻辑就在与之相关的线程中执行，相应的消息的走向也就在相关联的 MessageQueue 中。所以子线程切换到主线程是很自然的过程，并没有想象中的复杂。https://zhuanlan.zhihu.com/p/... 思路 2：Handler 利用线程封闭的 ThreadLocal 维持一个消息队列，Handler 的核心是通过这个消息队列来传递 Message，从而实现线程间通信。思路 3：实际线程间切换，就是通过线程间共享变量实现的。首现在一个线程中，new Handler() 必须先执行，Looper.prepare() 创先 looper，主线程没有手动调用 Looper.prepare()，是因为 app 启动时，在 ActivityThread main 主入口，执行了 Looper.prepareMainThread. public static void prepareMainLooper() { ... }

11 HashMap 原理

数据结构和算法思考 1. 为什么选择数组和链表结构？数组内存连续块分配，效率体现查询更快。HashMap 中用作查找数组桶的位置，利用元素的 key 的 hash 值对数组长度取模得到。链表效率体现增加和删除。HashMap 中链表是用来解决 hash 冲突，增删空间消耗平衡。扩展：为什么不是 ArrayList 而是使用 Node<K,V>[] tab？因为 ArrayList 的扩容机制是 1.5 倍扩容，而 HashMap 扩容是 2 的次幂。2.HashMap 出现线程问题 多线程扩容，引起的死循环问题（jdk1.8 中，死循环问题已经解决）。多线程 put 的时候可能导致元素丢失 put 非 null 元素后 get 出来的却是 null 3. 使用线程安全 Map HashMap 并不是线程安全，要实现线程安全可以用 Collections.synchronizedMap(m) 获取一个线程安全的 HashMap。CurrentHashMap 和 HashTable 是线程安全的。CurrentHashMap 使用分段锁技术，要操作节点先获取段锁，在修改节点。4.Android 提倡使用 ArrayMap ArrayMap 数据结构是两个数组，一个存放 hash 值，另一个存放 key 和 value。根据 key 的 hash 值利用二分查找在 hash 数组中找出 index。根据 index 在 key-value 数组中对应位置查找，如果不相等认为冲突了，会以 key 为中心，分别上下展开，逐一查找。优势，数据量少时（少于 1000）相比 HashMap 更节省内存。劣势，删除和插入时效率要比 HashMap 要低。

12 Android 打包流程: 熟悉 Android 打包编译的流程

- AAPT（Android Asset Packaging Tool）工具，Android 资源打包工具。会打包资源文件（res 文件夹下的文件），并生成 R.java 和 resources.arsc 文件。
- AIDL 工具会将所有的.aidl 文件编译成.java 文件。
- JAVAC 工具将 R.java、AIDL 接口生成的 java 文件、应用代码 java 文件编译成.class 文件。
- dex 脚本将很多.class 文件转换打包成一个.dex 文件。
- apkbuilder 脚本将资源文件和.dex 文件生成未签名的.apk 文件。
- jarsigner 对 apk 进行签名。

13 谈谈 Android 的 GC

Java 语言建立了垃圾收集机制，用以跟踪正在使用的对象和发现并回收不再使用 (引用) 的对象。该机制可以有效防范动态内存分配中可能发生的两个危险: 因内存垃圾过多而引发的内存耗尽，以及不恰当的内存释放所造成的内存非法引用。垃圾收集算法的核心思想是: 对虚拟机可用内存空间，即堆空间中的对象进行识别，如果对象正在被引用，那么称其为存活对象，反之，如果对象不再被引用，则为垃圾对象，可以回收其占据的空间，用于再分配。垃圾收集算法的选择和垃圾收集系统参数的合理调节直接影响着系统性能，因此需要开发人员做比较深入的了解。

14 怎样保证 App 不被杀死?

- 强烈建议不要这么做，不仅仅从用户角度考虑，作为 Android 开发者也有责任去维护 Android 的生态环境。当然从可行性讲，谷歌也不会让容易的实现。同时这样的 app 一般属于流氓应用
- 通常为了保证自己 app 避免被杀死，我们一般使用以下方法:
  - 1.Service 设置成 START$_{STICKY\ kill}$ 后会被重启 (等待 5 秒左右)，重传 Intent，保持与重启前一样
  - 2. 通过 startForeground 将进程设置为前台进程，做前台服务，优先级和前台应用一个级别，除非在系统内存非常缺，否则此进程不会被 kill
  - 3.. 双进程 Service: 让 2 个进程互相保护，其中一个 Service 被清理后，另外没被清理的进程可以立即重启进程
  - 4.QQ 黑科技: 在应用退到后台后，另起一个只有 1 像素的页面停留在桌面上，让自己保持前台状态，保护自己不被后台清理工具杀死
  - 5. 在已经 root 的设备下，修改相应的权限文件，将 App 伪装成系统级的应用 (Android4.0 系列的一个漏洞，已经确认可行)
  - 6.Android 系统中当前进程 (Process)fork 出来的子进程，被系统认为是两个不同的进程。当父进程被杀死的时候，子进程仍然可以存活，并不受影响。鉴于目前提到的在 Android-Service 层做双守护都会失败，我们可以 fork 出 c 进程多进程守护。死循环在那检查是否还存在，具体的思路如下 (Android5.0 以下可行)：
    - 1. 用 C 编写守护进程 (即子进程)，守护进程做的事情就是循环检查目标进程是否存在，不存在则启动它。
    - 2. 在 NDK 环境中将 1 中编写的 C 代码编译打包成可执行文件 (BUILD$_{EXECUTABLE}$)。
    - 3. 主进程启动时将守护进程放入私有目录下，赋予可执行权限，启动它即可。
  - 7 联系厂商，加入白名单

15 设计模式

Android 常用设计模式及源码使用单例模式初始化比较复杂，并且程序中只需要一个。避免重复创建消耗内存 Android 中获取 WindowManager 服务引用 WindowManager wm = (WindowManager)getSystemService(getApplication().WINDOW$_{SERVICE}$); l 另外一种不错实现单例的方式使用 eunm, public class Singleton { private static volatile Singleton s; private Singleton(){}; public static Singleton getInstance() { if(s == null) { synchronized (Singleton.class) { if(s == null) { s = new Singleton(); } } } return s; 创建者模式创建某对象时，需要设定很多的参数（通过 setter 方法），但是这些参数必须按照某个顺序设定 Android 中创建所有的 Dialog 中使用的 public class TestClient {

private int index; private String name; public TestClient() { this(new Builder()); } public TestClient(Builder builder){ this.index = builder.index; this.name = builder.name; } public static final class Builder { private int index; private String name; public Builder() { this.index = 1; this.name = "xxx"; } public Builder(TestClient testClient){ this.index = testClient.index; this.name = testClient.name; } public Builder setIndex(int index) { this.index = index; return this; } public Builder setName(String name) { this.name = name; return this; } public TestClient build(){ return new TestClient(this); } } } 原型模式工厂模式定义一个创建对象的工厂，根据不同传参创建不同的对象。Android 中 BitmapFactory 和 Iterator 根据循环对象不同返回不同的对象策略模式有一系列的算法，将算法封装起来（每个算法可以封装到不同的类中），各个算法之间可以替换，策略模式让算法独立于使用它的客户而独立变化 Android 中的时间插值器，可以使用不同的加速减速或者自定义加速器展示不同的动画效果责任链模式 B 命令模式命令模式将每个请求封装成一个对象，从而让用户使用不同的请求把客户端参数化；将请求进行排队或者记录请求日志，以及支持可撤销操作。Android 事件机制中，底层逻辑对事件的转发处理。每次的按键事件会被封装成 NotifyKeyArgs 对象，通过 InputDispatcher 封装具体的事件操作 / Runable 实现中封装我们需要的实现观察者模式 Java 的 Observable 类和 Observer 接口就是实现了观察者模式。一个 Observer 对象监视着一个 Observable 对象的变化，当 Observable 对象发生变化时，Observer 得到通知，就可以进行相应的工作。中介者模式在 Binder 机制中，即 ServiceManager 持有各种系统服务的引用，当我们需要获取系统的 Service 时，首先是向 ServiceManager 查询指定标示符对应的 Binder，再由 ServiceManager 返回 Binder 的引用。并且客户端和服务端之间的通信是通过 Binder 驱动来实现，这里的 ServiceManager 和 Binder 驱动就是中介者。代理模式给某一个对象提供一个代理，并由代理对象控制对原对象的引用 (, 静态代理和动态代理) 适配器模式把一个类的接口变换成客户端所期待的另一个接口，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。补充说明：例模式 //获取 WindowManager 服务引用 WindowManager wm = (WindowManager)getSystemService(getApplication().WINDOW$_{SERVICE}$); 基本所有的获取系统服务都是单例。Builder 模式一般常用于构建需要 3 个以上的参数。AlertDialog.Builer builder=new AlertDialog.Builder(context); builder.setIcon(R.drawable.icon) .setTitle("title") .setMessage("message") .setPositiveButton("Button1", new DialogInterface.OnclickListener(public void onClick(DialogInterface dialog,int whichButton){ setTitle("click"); } }) .create() .show(); 原型模式 Uri uri=Uri.parse("smsto:10086"); Intent shareIntent=new Intent(Intent.ACTION$_{SE}$ //克隆副本 Intent intent=(Intetn)shareIntent.clone(); startActivity(intent); 工厂模式 public Object getSystemService(String name) { if (getBaseContext() == null) { throw new IllegalStateException("System services not available to Activities before onCreate()"); } //. . . . . if (WINDOW$_{SERVICE}$.equals(name)) { return mWindowManager; } else if (SEARCH$_{SERVICE}$.equals(name)) { ensureSearchManager(); return mSearchManager; } //. . . . . return super.getSystemService(name); } 策略者模式根据是否是 v 包选择不同的构建 view 方案责任链模式 view 的点击事件，view 的 touch 事件等命令模式按键事件会被封装成 notifyKeyArgs 对象，通过 inputDispatcheri 封装具体事件操作观察者模式监听器这一类都是备忘录模式 activtity onSaveInstanceState 等迭代器模式数据库的 cursor，常用于查询代理模式 Binder 代理适配器模式 adapter 类装饰者模式 Rxjava2，或者 contextThemeWapper 等

16 跨平台

Flutter 和 React Native 不同主要在于 Flutter UI 是直接通过 skia 渲染的，而 React Native 是将 js 中的控件转化为原生控件，通过原生去渲染的。移动端跨平台开发的深度解析

17 网络

HTTP 请求方式和报文解析 1.HTTPS(Secure) 安全的 HTTP 协议 https://< 主机 >:<443>/< 路径 > 补充说明（其他版本）HTTPS 就是“安全版”的 HTTP, HTTPS = HTTP + SSL。HTTPS 相当于在应用层和 TCP 层之间加入了一个 SSL（或 TLS），SSL 层对从应用层收到的数据进行加密。TLS/SSL 中使用了 RSA 非对称加密，对称加密以及 HASH 算法。RSA 算法基于一个十分简单的数论事实：将两个大素数相乘十分容易，但那时想要对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥。1.1 加密模型「对称加密：加密与解密都使用同一个秘钥」。「非对称加密：公钥加密，私钥解密，并且公钥与私钥是拥有一定数学关系的一组秘钥」。「私钥：自己使用，不对外公开」。「公钥：给大家使用，对外公开」。1.2 数字证书签名校验数字证书格式证书格式、版本号证书序列号签名算法有效期对象名称对象公开秘钥 1.3 SSL(Secure Sockets Layer) 安全套接层「SSL 位于传输层与应用层之间，它是一个子层，作用主要有两点」：1)、「数据安全（保证数据不会被泄漏）与数据完整（保证数据不会被篡改）」。2)、「对数据进行加密后传输」。1.4 HTTPS 的通信过程 1)、「443 端口的 TCP 连接」。2)、「SSL 安全参数握手」。3)、「客户端发送数据」。4)、「服务端发送数据」。1.5 SSL(Secure Sockets Layer) 安全套接层握手过程 1)、生成随机数 1、2、3 的过程 image-20200527230427775 2)、双端根据随机数 1、2、3 与相同的算法生成对称秘钥进行加密通信 image-20200527230523455 「HTTPS 综合地运用了对称加密与非对称加密，在进行随机数校验的阶段是使用了非对称加密来进行通信的，然后等双方都确定了三个随机数之后，就可以使用相同的算法来生成对称秘钥进行加密通信了。HTTPS 的优势在于双端分别生成了秘钥，没有经过传输，减少了秘钥泄漏的可能性」。https 在项目的运用实践：Android HTTPS 自制证书实现双向认证 (OkHttp + Retrofit + Rxjava)

18 MVC MVP MVVM

19 java 基础

1.synchronized 的修饰对象当 synchronized 用来修饰静态方法或者类时，将会使得这个类的所有对象都是共享一把类锁，导致线程阻塞，所以这种写法一定要规避无论 synchronized 关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果 synchronized 作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

1. try{}catch{}finally 中的执行顺序?
2. 任何执行 try 或者 catch 中的 return 语句之前，都会先执行 finally 语句，如果 finally 存在的话。
3. 如果 finally 中有 return 语句，那么程序就 return 了，所以 finally 中的 return 是一定会被 return 的，
4. 编译器把 finally 中的 return 实现为一个 warning。

```
public class tmp {
    public static void main(String[] args) {
        System.out.println(test());;
```

```
    }
    static int test() {
        int x = 1;
        System.out.println("x 0: " + x);
        try {
            x++;
            System.out.println("x 1: " + x);
            return x;
        } finally {
            ++x;
            System.out.println("x 2: " + x);
        }
    }
}
// x 0: 1
// x 1: 2
// x 2: 3
```

- 说明：在 try 语句中，在执行 return 语句时，要返回的结果已经准备好了，就在此时，程序转到 finally 执行了。
- 在转去之前，try 中先把要返回的结果存放到不同于 x 的局部变量中去，执行完 finally 之后，在从中取出返回结果，
- 因此，即使 finally 中对变量 x 进行了改变，但是不会影响返回结果。
- 它应该使用栈保存返回值。
- JAVA 中的死锁
- JAVA 中的 ArrayList 是否是线程安全
- 为什么 ArrayList 线程不安全？不安全为什么要使用？如何解决线程不安全？
- 首先说一下什么是线程不安全：线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。线程不安全就是不提供数据访问保护，有可能出现多个线程先后更改数据造成所得到的数据是脏数据。List 接口下面有两个实现，一个是 ArrayList，另外一个是 vector。
- 从源码的角度来看，因为 Vector 的方法前加了，synchronized 关键字，也就是同步的意思，sun 公司希望 Vector 是线程安全的，而希望 arraylist 是高效的，缺点就是另外的优点。
- 说下原理：一个 ArrayList ，在添加一个元素的时候，它可能会有两步来完成：
  - 在 Items[Size] 的位置存放此元素；
  - 增大 Size 的值。
  - 在单线程运行的情况下，如果 Size = 0，添加一个元素后，此元素在位置 0，而且 Size=1；
  - 而如果是在多线程情况下，比如有两个线程，线程 A 先将元素存放在位置 0。但是此时 CPU 调度线程 A 暂停，线程 B 得到运行的机会。线程 B 也向此 ArrayList 添加元素，因为此时 Size 仍然等于 0 （注意哦，我们假设的是添加一个元素是要两个步骤哦，而线程 A 仅仅完成了步骤 1），所以线程 B 也将元素存放在位置 0。然后线程 A 和线程 B 都继续运行，都增加 Size 的值。

那好，现在我们来看看 ArrayList 的情况，元素实际上只有一个，存放在位置 0，而 Size 却等于 2。这就是“线程不安全”了。

- 不安全为什么要使用？
- 这个 ArrayList 比线程安全的 Vector 效率高。

- 如何解决线程不安全
  - 使用 synchronized 关键字，这个大家应该都很熟悉了，不解释了；
  - 使用 Collections.synchronizedList(); 使用方法如下：
    * 假如你创建的代码如下：

```
List<Map<String,Object>>data=new ArrayList<Map<String,Object>>();
```

- 那么为了解决这个线程安全问题你可以这么使用 Collections.synchronizedList()，如：

```
List<Map<String,Object>> data=Collections.synchronizedList(newArrayList<Map<String,Object>>());
```

- 其他的都没变，使用的方法也几乎与 ArrayList 一样，大家可以参考下 api 文档；
- 额外说下 ArrayList 与 LinkedList：这两个都是接口 List 下的一个实现，用法都一样，但用的场所的有点不同，ArrayList 适合于进行大量的随机访问的情况下使用，LinkedList 适合在表中进行插入、删除时使用，二者都是非线程安全，解决方法同上（为了避免线程安全，以上采取的方法，特别是第二种，其实是非常损耗性能的）。

原文链接：https://blog.csdn.net/qq_2808...

- JAVA 和 Vector 的区别
- 首先看这两类都实现 List 接口，而 List 接口一共有三个实现类，分别是 ArrayList、Vector 和 LinkedList。List 用于存放多个元素，能够维护元素的次序，并且允许元素的重复。3 个具体实现类的相关区别如下：
  - 1.ArrayList 是最常用的 List 实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要讲已经有数组的数据复制到新的存储空间中。当从 ArrayList 的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。
  - 2.Vector 与 ArrayList 一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写 Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问 ArrayList 慢。
  - 3.LinkedList 是用链表结构存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较慢。另外，他还提供了 List 接口中没有定义的方法，专门用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。
  - 4.vector 是线程（Thread）同步（Synchronized）的，所以它也是线程安全的，而 Arraylist 是线程异步（ASynchronized）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。
  - 5. 如果集合中的元素的数目大于目前集合数组的长度时，vector 增长率为目前数组长度的 100%, 而 arraylist 增长率为目前数组长度的 50%。如果在集合中使用数据量比较大的数据，用 vector 有一定的优势。
  - 6. 如果查找一个指定位置的数据，vector 和 arraylist 使用的时间是相同的，都是 0(1), 这个时候使用 vector 和 arraylist 都可以。而如果移动一个指定位置的数据花费的时间为 0(n-i)n 为总长度，这个时候就应该考虑到使用 Linkedlist, 因为它移动一个指定位置的数据所花费的时间为 0(1), 而查询一个指定位置的数据时花费的时间为 0(i)。
    * ArrayList 和 Vector 是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，

    - 都允许直接序号索引元素，但是插入数据要设计到数组元素移动等内存操作，所以索引数据快插入数据慢，
    - Vector 由于使用了 synchronized 方法（线程安全）所以性能上比 ArrayList 要差，LinkedList 使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快！
  - 7. 笼统来说：LinkedList：增删改快, ArrayList：查询快（有索引的存在）

4.synchronized 和 volatile 关键字的区别

- 1.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，需要从主存中读取;

- synchronized 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

- 2.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
- 3.volatile 仅能实现变量的修改可见性，不能保证原子性; 而 synchronized 则可以保证变量的修改可见性和原子性
- 4.volatile 不会造成线程的阻塞;synchronized 可能会造成线程的阻塞。
- 5.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

5.Java 中的自动装箱和自动拆箱

- 所以，当“==”运算符的两个操作数都是包装器类型的引用，则是比较指向的是否是同一个对象，而如果其中有一个操作数是表达式（即包含算术运算）则比较的是数值（即会触发自动拆箱的过程）。
- 通过上面的分析我们需要知道两点：
  - 1、什么时候会引发装箱和拆箱
  - 2、装箱操作会创建对象，频繁的装箱操作会消耗许多内存，影响性能，所以可以避免装箱的时候应该尽量避免。
- https://zhidao.baidu.com/ques...
- 为什么我们在 Java 中使用自动装箱和拆箱？

6.Java 中的乐观锁和悲观锁

- 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）。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。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，像数据库提供的类似于 write$_{condition}$ $_{Java\ java}$.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。

- 两种锁的使用场景
  - 从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行 retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

7. 抽象类和接口如何理解与记忆：1. 抽象类–> 像 xxx 一样接口–> 能 xxx 这样 2. 接口是设计的结果，抽象类是重构的结果。8.JAVA 内存模型 jmm 内存分配的概念：堆 heap：优点：运行时数据区，动态分配内存大小，有 gc;, 缺点:因为要在运行时动态分配，所以存取速度慢，对象存储在堆上，静态类型的变量跟着类的定义一起存储在堆上。栈 stack：存取速度快，仅次于寄存器，缺点：数据大小与生存期必须是确定的，缺乏灵活性，栈中主要存放基本类型变量（比如，int,shot,byte,char,double,foalt,boolean 和对象句柄），jmm 要求，调用栈和本地变量存放在线程栈上当一个线程可以访问一个对象时，也可以访问对象的成员变量，如果有两个线程访问对象的成员变量，则每个线程都有对象的成员变量的私有拷贝。处理器（cpu）：寄存器：每个 cpu 都包含一系列寄存器，他们是 cpu 的基础，寄存器执行的速度，远大于在主存上执行的速度 cpu 高速缓存：由于处理器与内存访问速度差距非常大，所以添加了读写速度尽可能接近处理器的高速缓存，来作为内存与处理器之间的缓冲，将数据读到缓存中，让运算快速进行，当运算结束，再从缓存同步到主存中，就无须等待缓慢的内存读写了。处理器访问缓存的速度快与访问主存的速度，但比访问内部寄存器的速度还是要慢点，每个 cpu 有一个 cpu 的缓存层，一个 cpu 含有多层缓存，，某一时刻，一个或者多个缓存行可能同时被读取到缓存取，也可能同时被刷新到主存中，同一时刻，可能存在多个操作，内存：一个计算机包含一个主存，所有 cpu 都可以访问主存，主存通常远大于 cpu 中的缓存，运作原理：通常，当一个 cpu 需要读取主存时，他会将主存的内容读取到缓存中，将缓存中的内容读取到内部寄存器中，在寄存器中执行操作，当 cpu 需要将结果回写到主存中时，他会将内部寄存器的值刷新到缓存中，然后会在某个时间点将值刷新回主存。原文链接：https://blog.csdn.net/wangnan... 9.GC Roots 如何确定？哪些对象可以作为 GC Roots？原文链接：https://blog.csdn.net/weixin_... 判断对象是否可以被回收之引用计数法: Java 中，引用和对象是有关联的。如果要操作对象则必须用引用进行。因此，很显然一个简单的办法是通过引用计数来判断一个对象是否可以回收。简单说，给对象中添加一个引用计数器，每当有一个地方引用它，计数器值加 1，每当有一个引用失效时，计数器值减 1。任何时刻计数器值为零的对象就是不可能再被使用的，那么这个对象就是可回收对象。那为什么主流的 Java 虚拟机里面都没有选用这种算法呢？其中最主要的原因是它很难解决对象之间相互循环引用的问题。判断对象是否可以被回收之枚举根节点可达性分析为了解决引用计数法的循环引用问题，Java 使用了可达性分析的方法。所谓"GC roots，或者说 tracing GC 的“根集合”就是一组必须活跃的引用。基本思路就是通过一系列名为”GCRoots”的对象作为起始点，从这个被称为 GC Roots 的对象开始向下搜索，如果一个对象到 GCRoots 没有任何引用链相连时，则说明此对象不可用。也即给定一个集合的引用作为根出发，通过引用关系遍历对象图，能被遍历到的（可到达的）对象就被判定为存活，没有被遍历到的就自然被判定为死亡。Java 中可以作为 GC Roots 的对象: 虚拟机栈（栈帧中的本地变量表）中引用的对象方法区中类静态属性引用的对象方法区中常量引用的对象本地方法栈中 JNI（即一般说的 native 方法）中引用的对象抽象类和接口的区别

抽象类与接口之间的关系

No.	区别点	抽象类	接口
1	定义	包含一个抽象方法的类	抽象方法和全局常量的集合
2	组成	构造方法、抽象方法、普通方法、常量、变量	常量、抽象方法
3	使用	子类继承抽象类（extends）	子类实现接口（implements）
4	关系	抽象类可以实现多个接口	接口不能继承抽象类，但允许继承多个接口
5	常见设计模式	模板设计	工厂设计、代理设计
6	对象	都通过对象的多态性产生实例化对象	
7	局限	抽象类有单继承的局限	接口没有此局限
8	实际	作为一个模板	是作为一个标准或是表示一种能力
9	选择	如果抽象类和接口都可以使用的话，优先使用接口，因为避免单继承的局限	
10	特殊	一个抽象类中可以包含多个接口，一个接口中可以包含多个抽象类	

四种引用的区别

引用类型	被垃圾回收时间	用途	生存时间
强引用	从来不会	对象的一般状态	JVM停止运行时终止
软引用	当内存不足时	对象缓存	内存不足时终止
弱引用	正常垃圾回收时	对象缓存	垃圾回收后终止
虚引用	正常垃圾回收时	跟踪对象的垃圾回收	垃圾回收后终止

Strong Reference Object StringBuilder builder = new StringBuilder(); This is the default type/class of Reference Object, if not differently specified: builder is a strong Reference Object. This kind of reference makes the referenced object not eligible for GC. That is, whenever an object is referenced by a chain of strong Reference Objects, it cannot be garbage collected. Weak Reference Object WeakReference<StringBuilder> weakBuilder = new WeakReference<StringBuilder>(builder); Weak Reference Objects are not the default type/class of Reference Object and to be used they should be explicitly specified like in the above example. This kind of reference makes the reference object eligible for GC. That is, in case the only reference reachable for the StringBuilder object in memory is, actually, the weak reference, then the GC is allowed to garbage collect the StringBuilder object. When an object in memory is reachable only by Weak Reference Objects, it becomes automatically eligible for GC. Levels of Weakness Two different levels of weakness can be enlisted: soft and phantom. A soft Reference Object is basically a weak Reference Object that remains in memory a bit more: normally, it resists GC cycle until no memory is available and there is risk of OutOfMemoryError (in that case, it can be removed). On the other hand, a phantom Reference Object is useful only to know exactly when an object has been effectively removed from memory: normally they are used to fix weird finalize() revival/resurrection behavior, since they actually do not return the object itself but only help in keeping track of their memory presence. Weak Reference Objects are ideal to implement cache modules. In fact, a sort of automatic eviction can be implemented by allowing the GC to clean up memory areas whenever objects/values are no longer reachable by strong references chain. An example is the WeakHashMap retaining weak keys.

- https://juejin.cn/post/7007326164268105741 这个有没有时间看再说