目录

[第五章 JVM（类的加载）、垃圾回收（GC） 3](#_Toc139963210)

[Java内存结构 3](#_Toc139963211)

[类的生命周期 3](#_Toc139963212)

[类加载器分类 3](#_Toc139963213)

[类加载器的加载模型 4](#_Toc139963214)

[判断是否是垃圾的算法 4](#_Toc139963215)

[回收前的挣扎方法以及要经历的过程 4](#_Toc139963216)

[Java的4种引用 5](#_Toc139963217)

[GC回收算法 5](#_Toc139963218)

[垃圾收集器 5](#_Toc139963219)

[MinorGC的过程 5](#_Toc139963220)

[对象的生命周期 6](#_Toc139963221)

[第六章 Java容器类 7](#_Toc139963222)

[容器类的分类以及同步集合、并发集合 7](#_Toc139963223)

[Map的3种遍历方式 8](#_Toc139963224)

[HashMap的底层 8](#_Toc139963225)

[哈希表的冲突处理方法 8](#_Toc139963226)

[HashMap的扩容 8](#_Toc139963227)

[HashTable、HashMap、HashSet区别（线程和NULL） 8](#_Toc139963228)

[ConcurrentHashMap用什么保证并发 9](#_Toc139963229)

[第七章 设计模式 9](#_Toc139963230)

[设计模式 9](#_Toc139963231)

[第九章 Java I/O与NIO 10](#_Toc139963232)

[Java IO字节和字符分别有什么类 10](#_Toc139963233)

[第十章 多线程 10](#_Toc139963234)

[同步、异步、互斥、并行、并发都是什么意思？ 10](#_Toc139963235)

[线程状态 10](#_Toc139963236)

[Thread.Start和run的区别 11](#_Toc139963237)

[创建线程 3 种方式 11](#_Toc139963238)

[实现互斥的两种锁： 11](#_Toc139963239)

[线程协作的方法 11](#_Toc139963240)

[FutureTask 12](#_Toc139963241)

[线程池的工作原理 12](#_Toc139963242)

[第一章 四大组件 12](#_Toc139963243)

[Activity生命周期七种方法 12](#_Toc139963244)

[Activity启动模式 12](#_Toc139963245)

[横竖屏切换Activity的生命周期的变化 12](#_Toc139963246)

[如果设置不销毁该如何保存数据 13](#_Toc139963247)

[Service两种启动方式和生命周期 13](#_Toc139963248)

[Service和Activity 通信 13](#_Toc139963249)

[ContentProvider的使用（一般用来访问系统数据库） 13](#_Toc139963250)

[广播分类 14](#_Toc139963251)

[广播注册方式 14](#_Toc139963252)

[第二章 Fragment 14](#_Toc139963253)

[Fragment生命周期 14](#_Toc139963254)

[ViewPager + Fragment的内存泄漏以及解决 15](#_Toc139963255)

[viewPager+ Fragment懒加载的3个条件 15](#_Toc139963256)

[Activity和Fragment通信方式（传递数据） 16](#_Toc139963257)

[EventBus是什么 16](#_Toc139963258)

[第三章 存储 16](#_Toc139963259)

[安卓的存储方式 16](#_Toc139963260)

[第四章 自定义组件、动画 17](#_Toc139963261)

[自定义组件绘制流程 17](#_Toc139963262)

[事件分发机制 17](#_Toc139963263)

[动画种类 19](#_Toc139963264)

[ListView 性能优化 19](#_Toc139963265)

[第五章 网络 之 计算机网络 19](#_Toc139963266)

[ISO/OSI 7层模型 19](#_Toc139963267)

[TCP和UDP的特点 19](#_Toc139963268)

[三次握手四次挥手 20](#_Toc139963269)

[Okhttp 基本底层原理 21](#_Toc139963270)

[第六章 图片 22](#_Toc139963271)

[Bitmap优化 22](#_Toc139963272)

[第七章 布局 22](#_Toc139963273)

[布局优化 22](#_Toc139963274)

[第八章 性能优化 22](#_Toc139963275)

[内存抖动意义？如何避免？ 22](#_Toc139963276)

[ANR & CRASH 产生的原因？如何解决？ 22](#_Toc139963277)

[内存泄漏的本质原因和4种容易发生的情况 23](#_Toc139963278)

[启动优化 23](#_Toc139963279)

[第十章 线程 / 进程 通信 24](#_Toc139963280)

[AsyncTask实现原理 24](#_Toc139963281)

[Handler作用 24](#_Toc139963282)

[Android 消息机制的工作原理 24](#_Toc139963283)

[第十二章 Android第三方库源码 25](#_Toc139963284)

[Glide 25](#_Toc139963285)

[第十三章 杂七杂八 25](#_Toc139963286)

[屏幕适配 25](#_Toc139963287)

[反射的三个方法 25](#_Toc139963288)

[动态代理 25](#_Toc139963289)

[自我介绍 25](#_Toc139963290)

[项目介绍 26](#_Toc139963291)

[MMKV存储框架 26](#_Toc139963292)

[MVVM 26](#_Toc139963293)

[View的Measure过程（默认） 27](#_Toc139963294)

[ViewGroup的Measure过程 28](#_Toc139963295)

[双重锁两层的作用和syncronize的作用 28](#_Toc139963296)

[Handler内部有什么方法，如何确保回调到这个消息 29](#_Toc139963297)

[RecyclerView的优化 30](#_Toc139963298)

[Kotlin协程 31](#_Toc139963299)

# 第五章 JVM（类的加载）、垃圾回收（GC）

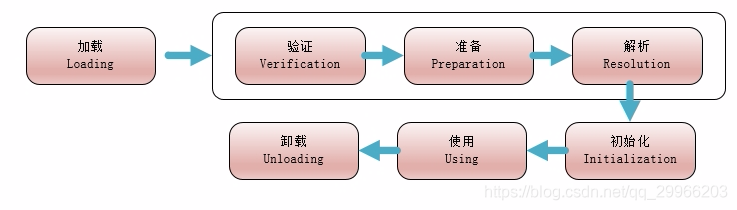
# Java内存结构

共享：堆内存（heap）、方法区（method）（存储类的信息，包含运行时常量池：存放编译器生成的各种字面量和符号引用）

线程私有：程序计数器、栈内存（stack）、本地方法栈（java中JNI调用）

# 类的生命周期

加载，验证，准备，解析，初始化，使用和卸载。验证，准备，解析3个部分统称为连接



加载: 生成Class对象并加载入内存作为数据的访问入口

验证：确保Class文件的字节流中包含的信息符合当前虚拟机的要求

准备：正式为类变量分配内存并设置类变量初始值

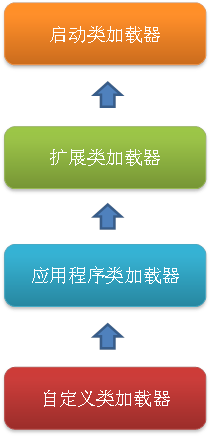
解析：虚拟机将常量池内的符号引用替换为直接引用的过程。

初始化：对类变量进行赋值及执行静态代码块

# 类加载器分类

启动类加载器

其他类加载器：扩展类加载器、应用程序类加载器、自定义类加载器



# 类加载器的加载模型

双亲委派模型

# 判断是否是垃圾的算法

1. **引用计数法：对象A的计数器的值为0，说明A没有引用，可以被回收**
2. **可达性分析算法：不可达节点则被认为是没有被引用到的节点**

# 回收前的挣扎方法以及要经历的过程

Finalize（）和两次标记

第一次标记：如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链则标记。

判断是否有finalize()方法，有则执行。

第二次标记：第二次标记如果还是不可达则标记并回收。

# Java的4种引用

强引用：不会被垃圾回收器回收

软引用：内存不够回收

弱引用：生存到下一次垃圾收集发生之前

虚引用：作用是收到通知

# GC回收算法

1. **标记清除法**
2. **标记压缩法**
3. **复制算法**

# 垃圾收集器

串行垃圾收集器

并行垃圾收集器

并发垃圾收集器

G1垃圾收集器取消了年轻代、老年代的物理划分，取而代之将堆划分为若干个区域（Regin），这些区域包含了有逻辑上的年轻代、老年代区域。

G1有三种GC Minor GC、Mixed GC、Full GC

# MinorGC的过程

Young 年轻代（复制算法）

1. 几乎所有新生成的对象首先是放在Eden区间。当Eden区间内存不足时，会发起一次GC。
2. 进行GC，Eden区所有存活对象都会复制到Survivor\_To区，Survivor\_From区中仍存活的对象根据他们的年龄决定去向，若年龄达到年龄阀值（MaxTenuringThreshold）的对象会移动到老年代，没有到达阀值的对象会复制到Survivor\_To区。
3. 清空Eden区和Survivor\_From区。然后交换Survivor\_To与Survivor\_From区。

若Survivor\_To区被填满（Survivor\_To区不足以存放Eden与Survivor\_From区中存活的对象时），此时所有存活的对象移动到老年代。

# 对象的生命周期

创建阶段(Created)

应用阶段(In Use)

不可见阶段(Invisible)

不可达阶段(Unreachable)

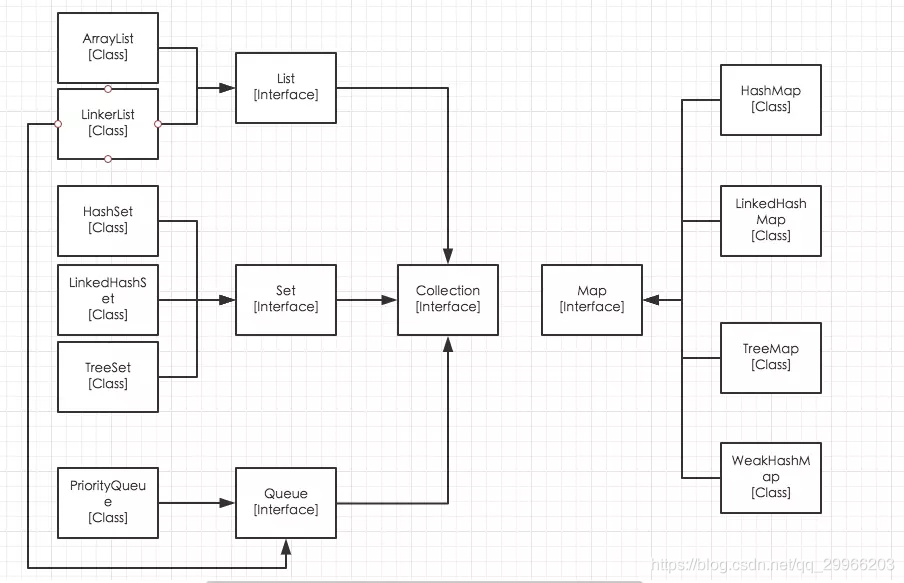
收集阶段(Collected)

终结阶段(Finalized)

对象空间重分配阶段(De-allocated)

# Java容器类

# 容器类的分类以及同步集合、并发集合



**Collection （Interface）：**Iterator迭代器遍历集合元素

**List （Interface）**ArrayList、LinkedList

**Set （Interface）**HashSet、LinkedSet、TreeSet

**Map （Interface）**

HashMap、LinkedHashMap、TreeMap、HashTable（线程安全）、WeakHashMap

同步集合类

Hashtable采用synchronized锁机制保证线程安全。

并发集合类

ConcurrentHashMap使用分离锁，16 个线程。

# Map的3种遍历方式

1. Set keySet() 获得所有key的集合然后通过key访问value
2. Collection values() 获得value的集合
3. Set< Map.Entry< K, V>> entrySet() 获得key-value键值对的集合

# [HashMap](https://so.csdn.net/so/search?q=HashMap&spm=1001.2101.3001.7020)的底层

数组+链表/红黑树，链表长度超过8个时会转化为红黑树

# 哈希表的冲突处理方法

开放定址法（线性探测再散列，二次探测再散列，伪随机探测再散列）

链地址法（HashMap头插法）

再哈希法（多个函数）

建立公共溢出区

# [HashMap](https://so.csdn.net/so/search?q=HashMap&spm=1001.2101.3001.7020)的扩容

超过了Load Factor（负载因子），bucket容量扩充为原来的两倍，然后对每个结点重新计算哈希值

# HashTable、HashMap、HashSet区别（线程和NULL）

线程安全：

是否可以插入NULL：

| **HashTable** | | **HashMap** | | **HashSet** |
| --- | --- | --- | --- | --- |
| 同步（synchronized） | | 不同步 | | 不同步 |
| 不可以 | 可以 | | 可以 | |

# ConcurrentHashMap用什么保证并发

利用CAS+Synchronized来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构

添加元素时首先会判断容器是否为空，

如果为空则使用 volatile 加 CAS 来初始化，

如果容器不为空，则根据存储的元素计算该位置是否为空。

如果根据存储的元素计算结果为空则利用 CAS 设置该节点；

如果根据存储的元素计算为空不为空，则使用 synchronized ，然后，遍历桶中的数据，并替换或新增节点到桶中，

最后再判断是否需要转为红黑树。这样就能保证并发访问时的线程安全了。

如果把上面的执行用一句话归纳的话，就相当于是ConcurrentHashMap通过对头结点加锁来保证线程安全的。

为空用CAS，不为空用synchronized

## ArrayList的动态扩容机制

按原容量的1.5倍进行扩容拷贝到新数组

# 第七章 设计模式

# 设计模式

创建型：单例（饿汉式、懒汉式、volatile+双重校验锁懒汉式）

结构型：适配器

行为型：责任链

# 第九章 Java I/O与NIO

# Java IO字节和字符分别有什么类

字符流

Reader，Writer

字节流

InputStream,OutputStream

# 多线程

# 同步、异步、互斥、并行、并发都是什么意思？

同步：进程必须顺序执行

异步：线程不需要顺序等待，可以去做别的事情

互斥：进程不能同时进行

并行：多处理器同时进行

并发：单处理器多任务同时进行

# 线程状态

1. 新建（New）
2. 可运行（Runnable）
3. 阻塞（Blocked）
4. 无限期等待（Waiting）：Object.wait() 、Thread.join()
5. 限期等待（Timed Waiting）：Thread.sleep() 、 Object.wait(xxx) 、Thread.join(xxx)
6. 死亡（Terminated）

# Thread.Start和run的区别

* Thread.start()：创建了新的线程，在新的线程中执行
* Thread.run()：在主线程中执行该方法，和调用普通方法一样

# 创建线程 3 种方式

实现Runnable接口

实现Callable接口-> FutureTask返回值封装

继承Thread类

# 实现互斥的两种锁：

## Synchronized 类（JVM）本质：

对象的监视器（monitor）的获取

## ReentrantLock 类（JUC）特点：

可公平的锁，需要主动释放

# 线程协作的方法

Join、wait() 、notify()、 notifyAll()、await() 、signal()、 signalAll()

join() 在线程中调用另一个线程的 join() 方法，会将当前线程挂起，而不是忙等待，直到目标线程结束。

wait() 、notify()、 notifyAll()（结合synchronized）

wait() 和 sleep() 的区别

wait() 是 Object 的方法，而 sleep() 是 Thread 的静态方法；

wait() 会释放锁，sleep() 不会。

await() 、signal()、 signalAll()（结合concurrent）

# FutureTask

FutureTask 可用于异步获取执行结果或取消执行任务的场景。当一个计算任务需要执行很长时间，那么就可以用 FutureTask 来封装这个任务，主线程在完成自己的任务之后再去获取结果。

# 线程池的工作原理

如果workerCount < 核心线程数，则创建并启动一个线程来执行新提交的任务；

如果workerCount >= 核心线程数，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中；

如果workerCount >= 核心线程数&& workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务；

如果workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满, 则根据拒绝策略来处理该任务, 默认的处理方式是直接抛异常。

# 四大组件

# Activity生命周期七种方法

# Activity启动模式

# 横竖屏切换Activity的生命周期的变化

->onPause->onSaveInstanceState->onStop->onDestroy->onCreate->onStart->onRestoreInstanceState->onResume

# 如果设置不销毁该如何保存数据

回调Activity的onConfigurationChanged方法

# Service两种启动方式和生命周期

startService()

onCreate-> onStartCommand-> onDestory

bindService()

onCreate-> onBind-> onUnBind-> onDestory

可以同时使用

# Service和Activity 通信

1. bindService + 回调函数
2. 广播（推荐LocalBroadcastManager）

# ContentProvider的使用（一般用来访问系统数据库）

ContentResolver

getContentResolver().insert(uri, values)

getContentResolver().query(uri, values)

// 由于短信数据库 系统通过ContentProvider 将接口暴露出来，因此可以直接通过ContentResolver 操作数据

getContentResolver().insert(uri, values)

getContentResolver().query(uri, values)

# 广播分类

普通广播

系统广播

有序广播

无序广播

全局广播

本地广播

# 广播注册方式

**静态注册**

动态注册

# 第二章 Fragment

# Fragment生命周期

onAttach

onCreate

onCreateView

onActivityCreated

onStart

onResume

onPause

onStop

onDestroyView

onDestroy

onDetach

# ViewPager + Fragment的内存泄漏以及解决

原因:

一般ViewPager + Fragment结合使用出现内存泄漏的原因可能用某个集合存储了Fragment的实例,导致当用户滑动ViewPager的时候，某一个Fragment即将面临销毁的时候，由于这个集合持有的它的引用，因此不能被回收掉,如果Fragment里面有大量的数据占据内存，有可能会导致OOM。

解决方法：

尽量不要使用集合来存储Fragment实例对象，除非你有良好的二次封装。再就是要做好每一页Fragment的数据缓存问题。

# viewPager+ Fragment懒加载的3个条件

ViewPager的缓存机制 —— 预加载

ViewPager会提前创建好左右一共3个Fragment。（其他的会被销毁）

缺点：完全加载所有的fragment就需要消耗大量的cpu资源，导致页面卡顿，性能低下。

优化：（数据）懒加载

旧版本：

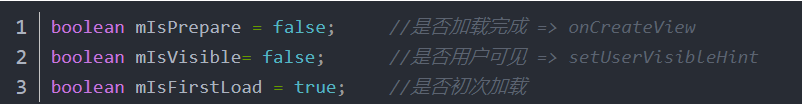
简单的说：就是把加载数据放到setUserVisibleHint方法中

setUserVisibleHint函数是游离在Fragment生命周期之外的，它的执行有可能早于onCreate和onCreateView，然而既然要时间数据的加载，就必须要在onCreateView创建完视图过后才能使用，不然就会返回空指针崩溃

实行懒加载必须满足的条件：

1. View视图加载完毕，即onCreateView（）执行完成
2. 当前Fragment可见，即setUserVisibleHint（）的参数为true
3. 初次加载，即防止多次滑动重复加载

对应三个标志参数



之后在对应的位置修改标志参数使之满足条件时加载数据即可

新版本：

ViewPager 2 底层是用 RecycleView 实现，已经默认实现懒加载

对于androidX，抛弃了setUserVisibleHint这个方法，底层使用LifeCycle来绑定fragment生命周期

# Activity和Fragment通信方式（传递数据）

1. 直接访问引用
2. Activity向Fragment通信——

若Fragment存在可通过manager调用FindFragmentById

若不存在可通过setArguments(bundle)/getArguments方法传递bundle参数

1. Fragment向Activity通信——回调函数
2. 第三方开源框架：EventBus
3. Fragment之间通信——以宿主Activity为桥梁

# EventBus是什么

EventBus（事件总线）是一个Android端优化的 发布/订阅 消息总线。简化活动Activity、碎片Fragment、进程Thread、服务Service等的通信方式。

# 第三章 存储

# 安卓的存储方式

Android本地存储方式有5种，分别是

SharedPreferences存储、

文件存储、

SQLite存储、

ContentProvider、

网络存储方式。

存储框架MMKV

# 第四章 自定义组件、动画

# 自定义组件绘制流程

Measure

Layout

Draw

绘制规则：

先绘制父类之后递归绘制子类

# 事件分发机制

Activity（Window） -> ViewGroup（容纳UI组件的容器，一组View的集合，如DecorView、Layout等） -> View（所有UI的基类）

Activity去分发，先给ViewGroup，如果ViewGroup要拦截就会调用onTouch方法，如果不拦截就会传递给子View，接着子View如果不消费就会按层级回传，最终又到达Activity。

onTouch、onTouchEvent （和onClick） 有什么区别

这两个方法都在View.dispatchTouchEvent()中调用

滚动事件onScroll与点击事件onClick等等均基于onTouchEvent

onTouch优先级比onTouchEvent优先级高，onTouchEvent中performClick是onClick的入口方法

onTouch>onTouchEvent>onClick



ViewGroup 想把自己分发给自己的onTouchEvent，需要拦截器onInterceptTouchEvent方法return true 把事件拦截下来。

View没有拦截器，它默认拦截

其实分发则对应递归，是向下遍历的过程，为了找到合适的child view进行消费；而消费则是回溯，是向上反馈的过程，决定了parent view是否可以选择进行消费。因此，消费是child view优先的。打个不太恰当的比喻，总当孩子说不吃时，父母才会去吃。当然父母也应该有自己的享受，这就对应了拦截

（1）dispatchTouchEvent 内部调用下面两个

用来进行事件的分发。如果事件能够传递给当前View,那么此方法一定会被调用，返回结果受当前View的onTouchEvent和下级的dispatchTouchEvent方法影响，表示是否消耗此事件。

（2）onInterceptTouchEvent 拦截器

在上述方法dispatchTouchEvent内部调用，用来判断是否拦截某个事件，返回结果表示是否拦截当前事件。如果当前View拦截了某个事件，则交给onTouchEvent继续处理。并且同一个事件序列当中，此方法不会被再次调用。

（3）onTouchEvent 消耗器

同样也会在dispatchTouchEvent内部调用，用来处理Touch事件，返回结果表示是否消耗当前事件，如果不消耗，则在同一个事件序列中，当前View无法再次接收到事件。

# 动画种类

补间动画

逐帧动画

属性动画

# ListView 性能优化

Bitmap优化

（1）用软引用存储图片信息

（2）图片压缩

（3）三级缓存

# 第五章 网络 之 计算机网络

# ISO/OSI 7层模型

物理层

数据链路层

网络层

传输层

会话层

表示层

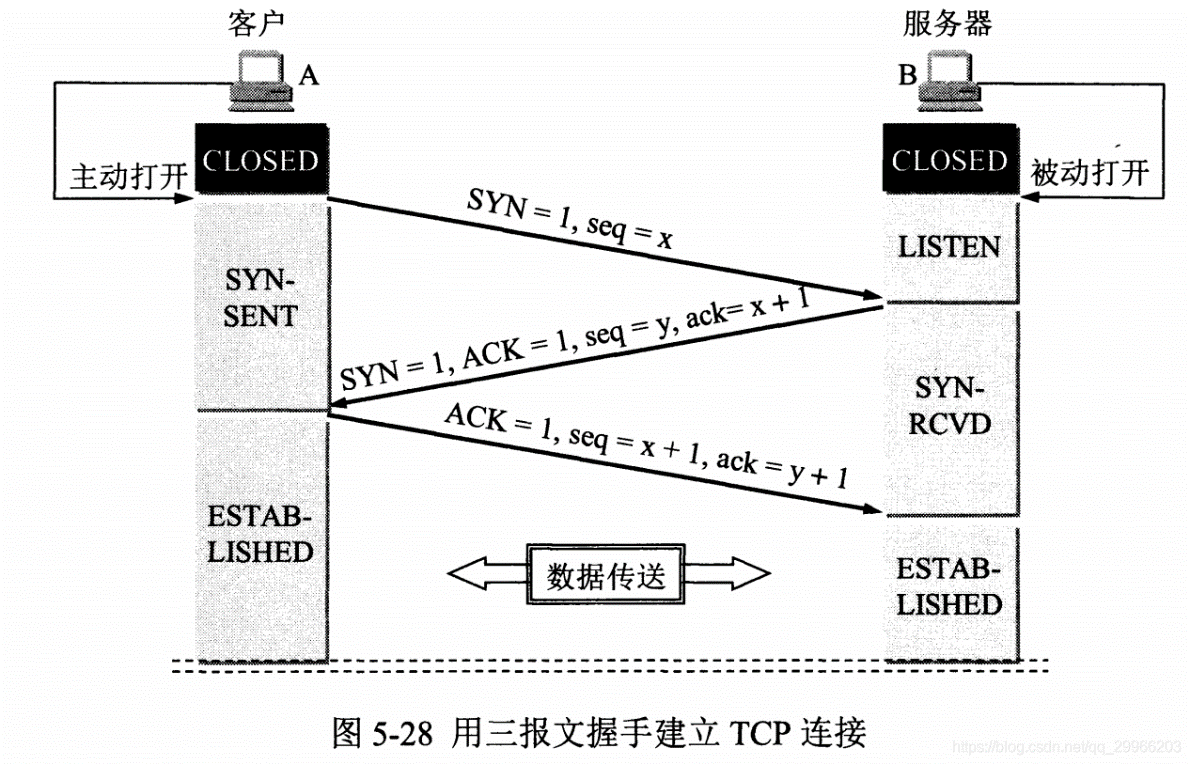
应用层

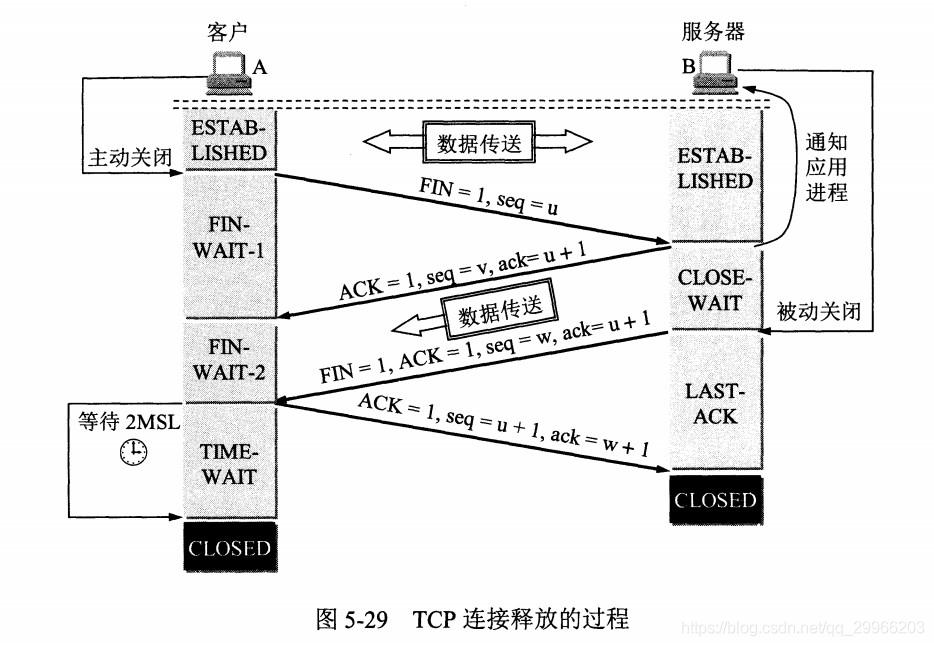
# TCP和UDP的特点

UDP无连接的不可靠的

TCP面向连接的可靠的

# 三次握手四次挥手





**第五章 网络 之 Android网络知识&框架**

# Okhttp 基本底层原理

OkHttp 主要是通过 5 个[拦截器](https://so.csdn.net/so/search?q=%E6%8B%A6%E6%88%AA%E5%99%A8&spm=1001.2101.3001.7020" \t "_blank)和1个线程池中的3个双端队列（2 个异步队列，1 个同步队列）工作。内部实现通过一个责任链模式完成，将网络请求的各个阶段封装到各个链条中，实现了各层的解耦。

OkHttp 的底层是通过 Socket 发送 HTTP 请求与接受响应，但是 OkHttp 实现了连接池的概念，即对于同一主机的多个请求，可以公用一个 Socket 连接。

实现流程：

1. OkhttpClient 负责为Request 创建 Call；
2. RealCall 为Call的具体实现，其enqueue() 异步请求接口通过Dispatcher()调度器利用ExcutorService实现，而最终进行网络请求时和同步的execute()接口一致，都是通过 getResponseWithInterceptorChain() 函数实现
3. getResponseWithInterceptorChain() 中利用 Interceptor 链条，责任链模式 分层实现缓存、透明压缩、网络 IO 等功能；最终将响应数据返回给用户。

Dispatcher类：  
Interceptor类：

Dispatcher类：

Dispatcher分发器，通过维护一个线程池executorService来分发Request。  
线程池内部维护着三个队列：同步请求队列 runningSyncCalls、异步请求队列 runningAsyncCalls、异步缓存队列 readyAsyncCalls，整体可以参照生产者消费者模式来理解：Dispatcher是生产者，executorService是消费者池，runningSyncCalls、runningAsyncCalls和readyAsyncCalls是消费者，用来消费请求Call。

Interceptor类：

官网：拦截器是Okhttp中提供的一种强大机制，它可以实现网络监听、请求、以及响应重写、请求失败重试等功能。  
RetryAndFollowUpInterceptor、BridgeIntercept、CacheIntercept、ConnectIntercept、CallServerIntercept

RetryAndFollowUpInterceptor：重试和失败重定向拦截器，负责监控请求是否失败

BridgeInterceptor：桥接拦截器，添加请求头、cookie信息

CacheInterceptor：缓存拦截器，命中缓存则不会发起网络请求。

ConnectInterceptor：连接拦截器，维护一个连接池，负责连接复用，创建和释放连接以及创建连接上的socket流。

CallServerInterceptor：请求服务器拦截器，发起网络请求。将 request写进 IO 流当中，并且从 IO 流中读取响应 Response

# 第六章 图片

# Bitmap优化

1. 使用完毕后释放图片资源
2. 设置图片缓存——三级缓存机制

内存缓存

文件缓存

联网加载

# 第七章 布局

# 布局优化

1. 善用相对布局RelativeLayout降低LinearLayout层级
2. 使用抽象布局标签include、merge、ViewStub（延迟加载布局，适用于布局复杂却很少用的布局）
3. 使用Android最新的布局方式ConstaintLayout

# 第八章 性能优化

# 内存抖动意义？如何避免？

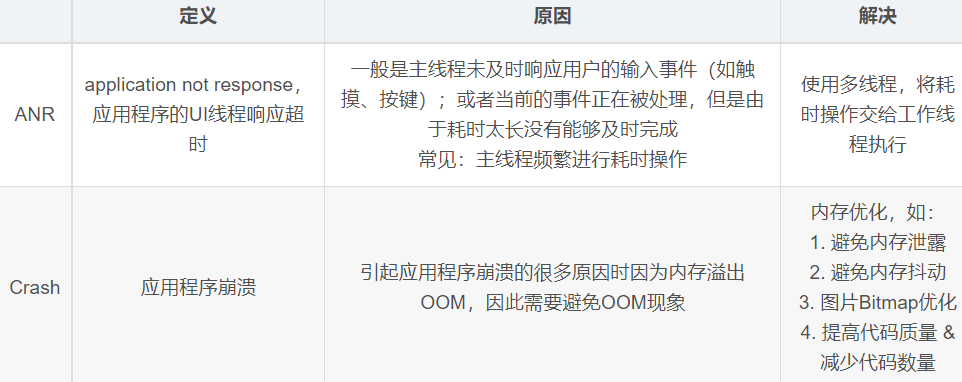
内存大小不断浮动的现象

尽量避免频繁创建大量、临时的小对象（导致内存碎片，不连续无法分配视为内存不足）

# ANR & CRASH 产生的原因？如何解决？

ANR：响应超时，耗时操作

Crash：崩溃，内存溢出OOM



# 内存泄漏的本质原因和4种容易发生的情况

持有引用的对象的生命周期>被引用的对象的生命周期

1. 集合类：回收集合元素
2. Static关键字修饰的成员变量：避免Static引用过多实例
3. 非静态内部类/匿名类：使用静态内部类
4. 资源使用后未关闭：关闭资源对象

# 启动优化

黑白屏优化

系统在启动Activity的setContentView之前绘制窗体，此时布局资源还未加载，于是使用了默认的背景色。

解决：把启动图bg\_splash设置为窗体背景，避免刚刚启动App的时候出现，黑/白屏

onCreate优化

onCreate()耗时长会影响应用程序布局绘制的时间。因此应该减少onCreate工作量。

一般重写Application，在onCreate()方法中做一些初始化操作（如第三方SDK配置），可以将这些较大的第三方库通过开启一个异步线程中进行初始化。

# 第十章 线程 / 进程 通信

# AsyncTask实现原理

一个Android已封装好的轻量级异步类，属于抽象类

2个线程池+Handler

任务队列线程池

执行线程池

内部Handler

# Handler作用

一种Android消息传递/异步通信机制

将工作线程中需更新UI的操作信息 传递到 UI主线程，保证多线程并发更新UI时 线程安全，实现异步消息的处理

# Android 消息机制的工作原理

异步通信准备 → 消息发送 → 消息循环 → 消息处理

创建 循环器Looper 对象、消息队列MessageQueue 对象、Handler 对象

通过Handler 发送消息Message 到消息队列MessageQueue中

Looper循环取出消息队列MessageQueue中的消息Message并发送给创建消息的处理者Handler

消息处理者Handler 接受 Looper 发送过来的消息Message，并根据Messge进行UI操作

Looper：

在一个线程中，如果存在Looper对象，则必定存在MessageQueue对象，并且只存在一个Looper对象和一个MessageQueue对象。在Android系统中，除了主线程有默认的Looper对象，其它线程默认是没有Looper对象。如果想让我们新创建的线程拥有Looper对象时，我们首先应调用Looper.prepare()方法，然后再调用Looper.loop()方法

# 第十二章 Android第三方库源码

# Glide

Glide，一个被google所推荐的图片加载库，这个库的主旨就在于让图片加载变的流畅。

根据Activity/Fragment的生命周期自动管理请求。

# 第十三章 杂七杂八

# 屏幕适配

根据 dp和px的转换公式 ：px = dp \* density ，可通过修改density保证所有设备计算出的px值满足该设备的要求。

# 反射的三个方法

Class.forName(String className)

Person.class

person.getClass()

# 动态代理

代理类是在运行时生成的。也就是说 Java 编译完之后并没有实际的 class 文件，而是在运行时动态生成的类字节码，并加载到JVM中。

接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）

# 自我介绍

面试官您好，我叫黄创恒，读于广东外语外贸大学，在学校学习的专业是软件工程，主修课程是java面向对象程序设计、数据结构、算法，成绩良好。我非常热爱手机开发，所以在大学时有幸加入了学校社团的安卓部门，接触了安卓的开发，这也让我走上了安卓开发的路。我熟悉Java和kotlin，对java知识总体有一定了解，包括：java泛型、注解、异常、反射、集合、多线程。对虚拟机的底层也有一些了解。包括：（类加载机制）、JVM内存模型、java GC。我在广州小迈科技有限公司实习了3个月，这段经历让我接触到真正的安卓开发工作，了解了一个项目从构建到上线的过程，也给我积累了很多开发经验，锻炼了我的团队合作能力，并让我养成了记录学习的好习惯。如果我有机会通过面试进入工作的话，我会在工作中努力学习、成长，为自己也为公司带来更多的价值。

# 项目介绍

我在实习期间主要负责的项目是天气app的日常开发。它基于MVVM框架采用组件化

开发。

我的主要工作是负责用kotlin语言进行天气真假页面的开发。主要用到的技术有：

tabLayout+viewPager+fragment实现页面的滑动切换、okhttp3获取网络数据、GSON解析JSON串、MMKV存储框架、自定义控件实现折线图等功能以及使用git来管理项目和合并项目，用jenkins工具来发布。

另外在性能优化上我也会注意内存泄漏的情况，在引用图片的时候会选用webp格式的图片并进行压缩，另外我也对JVM内存模型以及GC算法有一些了解。

# MMKV存储框架

是基于 mmap 内存映射的 key-value 组件，底层序列化/反序列化使用 protobuf 实现

Protobuf和Json类似但比Json有更高的转化率和空间效率

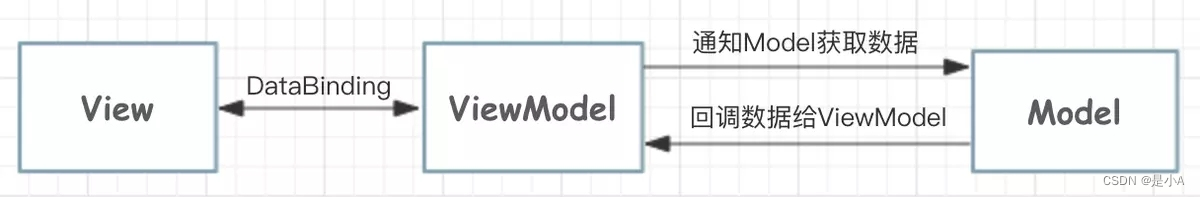
# MVVM

Model-View-ViewModel

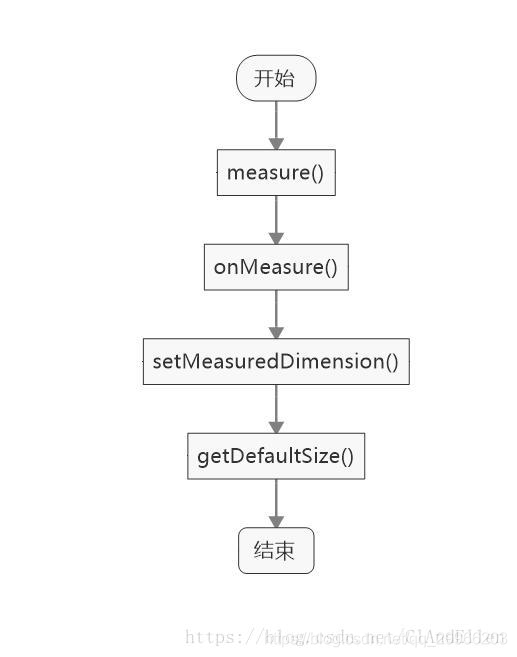
M——Model（模型）数据源（本地数据和网络数据等）实体模型，定义实体类，获取业务数据模型，如通过数据库或者网络来操作数据等

V——View（视图）xml、Activity、Fragment、Adapter和View等，主要进行控件的初始化设置

VM——ViewModel（控制器）：ViewModel 与 Model 直接交互处理数据以及逻辑处理，通过DataBinding将数据变化反应给View



# View的Measure过程（默认）



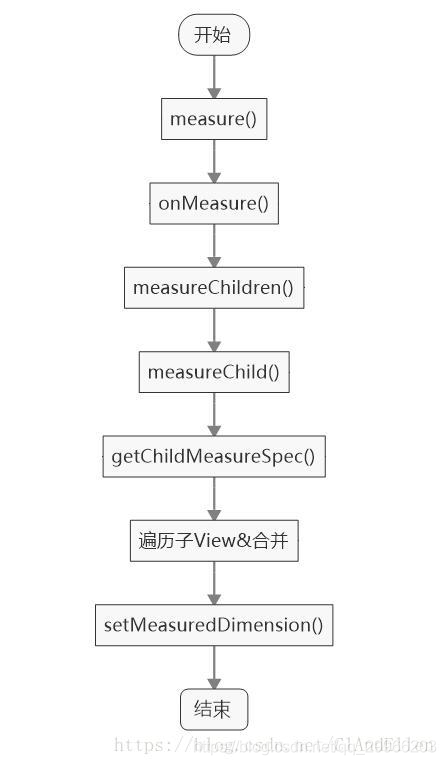
measure():基本测量逻辑的判断。

onMeasure():进行测量，会调用setMeasuredDimension（）方法

getDefaultSize():根据View宽/高的测量规格计算View的宽/高。

setMeasuredDimension():存储测量后的宽和高。值通过getDefaultSize（）中来

# ViewGroup的Measure过程



measure():基本测量逻辑的判断。

onMeasure():遍历所有的子View进行测量，调用measureChildren()方法。

measureChildren():遍历子View并对子View进行测量，后续会调用measureChild()方法。

measureChild():计算出单个子View的MeasureSpec，通过调用getChildMeasureSpce()方法实现，调用每个子View的measure()方法进行测量。

getChildMeasureSpec():计算出子View的MeasureSpec。

遍历子View并合并最终计算出ViewGroup的尺寸

setMeasuredDimension():存储测量后的宽和高。

# 双重锁两层的作用和syncronize的作用

if1，提高效率

Synchronize保证线程安全

If2，防止多次创建单例

校验锁1：第1个if

// 作用：若单例已创建，则直接返回已创建的单例，无需再执行加锁操作

// 即直接跳到执行 return ourInstance

校验锁2：第2个 if

// 作用：防止多次创建单例问题

// 原理

// 1. 线程A调用newInstance()，当运行到②位置时，此时线程B也调用了newInstance()

// 2. 因线程A并没有执行instance = new Singleton();，此时instance仍为空，因此线程B能突破第1层 if 判断，运行到①位置等待synchronized中的A线程执行完毕

// 3. 当线程A释放同步锁时，单例已创建，即instance已非空

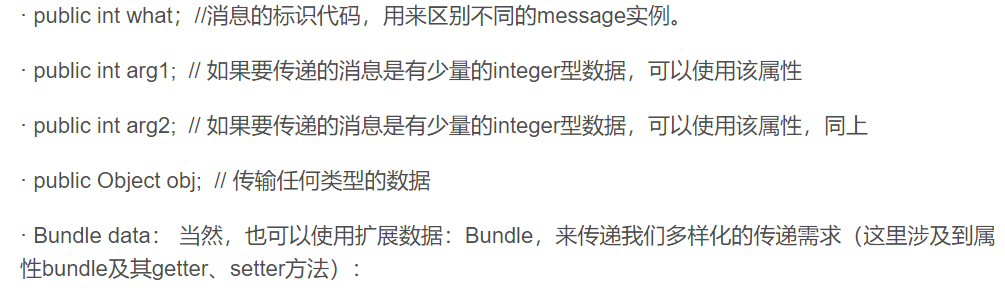
// 4. 此时线程B 从①开始执行到位置②。此时第2层 if 判断 = 为空（单例已创建），因此也不会创建多余的实例

# Handler内部有什么方法，如何确保回调到这个消息

1 sendMessage，post，handleMessage。

2使用Message.what属性标识handler实例

Message常用属性：



1. handler的作用无非就是发送消息、切换线程、处理消息。

sendMessage，post，handleMessage。

2. 在使用中，我们在哪个线程中创建了handler实例，哪个线程就是这个handler的“载体”、消息的接收端：在其他线程调用本线程的handler的sendMessage发送消息，那这条消息都会在本线程重写的方法handleMessage中接收到。

3. handler接受消息端是线程独立的，只有实例化本Handler的线程接受消息；但handler发送消息端多线程共享的：即拥有本Handler的引用的线程都可以作为消息的发送端，调用handler的发送消息方法来发送消息。所以，handler不是线程独立的。

所以，Handler需要一个独立存在于线程内部且私有使用的类来帮助它接受消息！

对，就是Looper！由此，Looper的作用就是辅助Handler接收消息且独立于线程内部。

那么问题又来了，如果同时有多个线程通过handler的引用向该handler发送消息，那么Looper怎么同时处理这些消息呢？

于是，为了防止多线程同时发消息Looper忙不过来，又设计了一个MessageQueue类，以队列的方式保存着Looper从其他线程接收到且未来得及发送给handler处理的消息，这样，Looper就可以一个个的有序的从MessageQueue中取出消息处理了。

可见，设计MessageQueue是为Looper服务的，Looper是线程独立的，那么MessageQueue也是线程独立的。

1.一个线程可以有多个Handler，而对于Handler来说，一个Handler只能实例化在某一个线程中，但其引用可以在任意多个线程（即前面说到的消息的发送端有多个，接收端只有一个）。

2. 一个线程只有一个looper，一个looper又唯一对应一个MessageQueue。

# RecyclerView的优化

现在能讲出来的：

1. 增加缓存数量
2. 减少布局层级
3. 使用局部刷新代替notifyDataSetChanged
4. 数据处理与视图绑定分离
5. 不要在onBindViewHolder中设置点击事件

https://blog.csdn.net/hxl517116279/article/details/107058425/

RecyclerView性能优化的本质就是针对onCreateViewHolder和onBindViewHolder的优化

1.减少onCreateViewHolder调用次数

1.1 两个数据源大部分相似时使用swapAdapter代替setAdapter

1.2 共用回收池：多个RecyclerView,如果使用同一个Adapter

1.3 增加RecycledViewPool缓存数量

2.减少onCreateViewHolder执行时间

2.1 减少item的过度绘制：减少布局层级，尽量少的布局嵌套，尽量少的控件

* 1. Prefetch预取

3. 减少onBindViewHolder调用次数

3.1 使用局部刷新：替代notifyDataSetChanged，达到局部刷新的目的

3.2 使用DiffUtil去局部刷新数据

3.3 视情况使用setItemViewCacheSize(size)来加大RecyclerView缓存数目，用空间换取时间提高流畅度

4.减少onBindViewHolder执行时间

4.1 数据处理与视图绑定分离

4.2 有大量图片时，滚动时停止加载图片，停止后再去加载图片：

https://www.jianshu.com/p/a8621f917407

4.3 使用setHasFixedSize避免requestLayout

4.4 不要在onBindViewHolder中设置点击事件

可以采取复用OnClickListener对象，然后在onBindViewHolder()方法中通过setTag(position) 和getTag() 的方式，来传递点击事件的position给listener。

5.1 对于RecyclerView，如果不需要动画，就把item动画取消

5.2 使用getExtraLayoutSpace为LayoutManager设置更多的预留空间

6.1 优化解耦 RecyclerView.Adapter

# Kotlin协程

https://blog.csdn.net/qq\_41811862/article/details/120808850

它其实和线程是有点类似的，可以简单地将它理解成一种轻量级的线程

线程是非常重量级的，它需要依靠操作系统的调度才能实现不同线程之间的切换。而使用协程却可以仅在编程语言的层面就能实现不同协程之间的切换，从而大大提升了并发编程的运行效率。

GlobalScope.launch函数创建的永远是顶层协程

launch函数

okhttp的使用enqueue和execute？

适配屏幕

多线程的普通使用：run和start等

多线程和主线程的切换

算法

ANR的日志在哪个路径

上下文的理解

# 线程池的参数



# Runnable接口和callable接口的区别

1. 方法不同

Runnable接口只有一个run()方法，该方法不返回任何值，因此无法抛出任何checked Exception。

Callable接口则有一个call()方法，它可以返回一个值，并且可以抛出一个checked Exception。

泛型的TE等参数

3. 异常处理不同

在Runnable中，我们无法对run()方法抛出的异常进行任何处理。

但在Callable中，自定义的call()方法可以抛出一个checked Exception，并由其执行者Handler进行捕获并处理。

小程序的刷新方法

RecyclerView的两层缓存机制

Kotlin和java的区别