ThreadLocal的理解

**Thread同步机制的比较**

　　ThreadLocal和线程同步机制相比有什么优势呢？ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。

　　在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

　　而ThreadLocal则从另一个角度来解决多线程的并发访问。ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

　　由于ThreadLocal中可以持有任何类型的对象，低版本JDK所提供的get()返回的是Object对象，需要强制类型转换。但JDK 5.0通过泛型很好的解决了这个问题，在一定程度地简化ThreadLocal的使用，代码清单 9 2就使用了JDK 5.0新的ThreadLocal<T>版本。

　　概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响。

Spring使用ThreadLocal解决线程安全问题我们知道在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean（如RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder等）中非线程安全状态采用ThreadLocal进行处理，让它们也成为线程安全的状态，因为有状态的Bean就可以在多线程中共享了。

　ThreadLocal是解决线程安全问题一个很好的思路，它通过为每个线程提供一个独立的变量副本解决了变量并发访问的冲突问题。在很多情况下，ThreadLocal比直接使用synchronized同步机制解决线程安全问题更简单，更方便，且结果程序拥有更高的并发性。

# 内存管理的细节

* **5.1内存回收**

在Android的高级系统版本里面针对Heap空间有一个Generational Heap Memory的模型，最近分配的对象会存放在Young Generation区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到Old Generation，最后累积一定时间再移动到Permanent Generation区域。系统会根据内存中不同的内存数据类型分别执行不同的gc操作。例如，刚分配到Young Generation区域的对象通常更容易被销毁回收，同时在Young Generation区域的gc操作速度会比Old Generation区域的gc操作速度更快。

* **5.2共享内存**

1. Android应用的进程都是从一个叫做Zygote的进程fork出来的。Zygote进程在系统启动并且载入通用的framework的代码与资源之后开始启动。为了启动一个新的程序进程，系统会fork Zygote进程生成一个新的进程，然后在新的进程中加载并运行应用程序的代码。这使得大多数的RAM pages被用来分配给framework的代码，同时使得RAM资源能够在应用的所有进程之间进行共享。
2. 大多数static的数据被mmapped到一个进程中。这不仅仅使得同样的数据能够在进程间进行共享，而且使得它能够在需要的时候被paged out。常见的static数据包括Dalvik Code，app resources，so文件等。
3. 大多数情况下，Android通过显式的分配共享内存区域(例如ashmem或者gralloc)来实现动态RAM区域能够在不同进程之间进行共享的机制。例如，Window Surface在App与Screen Compositor之间使用共享的内存，Cursor Buffers在Content Provider与Clients之间共享内存。

* **5.3分配与回收内存**

1. 每一个进程的Dalvik heap都反映了使用内存的占用范围。这就是通常逻辑意义上提到的Dalvik Heap Size，它可以随着需要进行增长，但是增长行为会有一个系统为它设定的上限。
2. 逻辑上讲的Heap Size和实际物理意义上使用的内存大小是不对等的，Proportional Set Size(PSS)记录了应用程序自身占用以及和其他进程进行共享的内存。

* **5.4限制应用的内存**

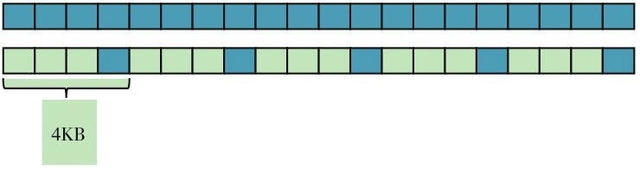
1. 为了整个Android系统的内存控制需要，Android系统为每一个应用程序都设置了一个硬性的Dalvik Heap Size最大限制阈值，这个阈值在不同的设备上会因为RAM大小不同而各有差异。如果你的应用占用内存空间已经接近这个阈值，此时再尝试分配内存的话，很容易引起OutOfMemoryError的错误。
2. ActivityManager.getMemoryClass()可以用来查询当前应用的Heap Size阈值，这个方法会返回一个整数，表明你的应用的Heap Size阈值是多少Mb(megabates)。

* **5.5应用切换**

1. Android系统并不会在用户切换应用的时候做交换内存的操作。Android会把那些不包含Foreground组件的应用进程放到LRU Cache中。例如，当用户开始启动了一个应用，系统会为它创建了一个进程，但是当用户离开这个应用，此进程并不会立即被销毁，而是会被放到系统的Cache当中，如果用户后来再切换回到这个应用，此进程就能够被马上完整的恢复，从而实现应用的快速切换。
2. 如果你的应用中有一个被缓存的进程，这个进程会占用一定的内存空间，它会对系统的整体性能有影响。因此当系统开始进入Low Memory的状态时，它会由系统根据LRU的规则与应用的优先级，内存占用情况以及其他因素的影响综合评估之后决定是否被杀掉。

**需要特别注意的：**

* 在Dalvik下，大部分Davik采取的都是标记-清理回收算法，而且**具体使用什么算法是在编译期决定的，无法在运行的时候动态更换。标记-清理回收算法无法对Heap中空闲内存区域做碎片整理。系统仅仅会在新的内存分配之前判断Heap的尾端剩余空间是否足够，如果空间不够会触发gc操作，从而腾出更多空闲的内存空间；这样内存空洞就产生了。**



如上图所示，**第一行，在开始阶段，内存分配较满；第二行，经过GC之后，大部分对象被释放。此时可能产生的问题是，因为没有内存整理功能，整个页面的4KB内存（内存分配的最小单位是页面，通常为4KB）可能只有一个小对象，但是统计PrivateDirty/Pss时还是按照4KB计算。所以对于Dalvik虚拟机的手机来说，我们首先要尽量避免掉频繁生成很多临时小变量（比如说：getView, onDraw等函数中new对象），另一个又要尽量去避免产生很多长生命周期的大对象。**

* **ART在GC上不像Dalvik仅有一种回收算法，ART在不同的情况下会选择不同的回收算法。应用程序在前台运行时，响应性是最重要的，因此也要求执行的GC是高效的。相反，应用程序在后台运行时，响应性不是最重要的，这时候就适合用来解决堆的内存碎片问题。因此，Mark-Sweep GC适合作为Foreground GC，而Mark-Compact GC适合作为Background GC。**由于有Compact的能力存在，内存碎片在ART上可以很好的被避免，这个也是ART一个很好的能力。