**[成都市天府软件园]**

**[tsotumu@163.com]**

**2018年8月22日**



**[此文档参考其他各种资料（包括CSDN.com、CNBLOG.com、阿里开发手册等），旨在提升开发效率（速度和质量）。资料中涉及广泛使用的编码规约及常用容器比较、多线程、文件操作、内存操作、分布式、调试、测试、Java知识扩展等。]**

[Java高效编程]

# **Java高效编程**

版本说明

|  |  |  |  |
| --- | --- | --- | --- |
| 版本号 | 作者 | 更新日期 | 备注 |
| 1.0 | tsotumu@163.com |  | 添加Java编码常识（编码规约及编码技巧） |

内容说明：

此文档参考其他各种资料（包括CSDN.com、CNBLOG.com、阿里开发手册等），旨在提升开发效率（速度和质量）。资料中涉及广泛使用的编码规约及常用容器比较、多线程、文件操作、内存操作、分布式、调试、测试、Java知识扩展等。

目录

**[第一章 设计模式](#_Toc32475_WPSOffice_Level1)** **[4](#_Toc32475_WPSOffice_Level1)**

[（一） 类的设计](#_Toc30454_WPSOffice_Level2) [4](#_Toc30454_WPSOffice_Level2)

[（二） 解耦实现高效](#_Toc23143_WPSOffice_Level2) [4](#_Toc23143_WPSOffice_Level2)

**[第二章 关于Object](#_Toc30454_WPSOffice_Level1)** **[4](#_Toc30454_WPSOffice_Level1)**

[（一） Class](#_Toc13718_WPSOffice_Level2) [4](#_Toc13718_WPSOffice_Level2)

[（二） hashCode](#_Toc28463_WPSOffice_Level2) [4](#_Toc28463_WPSOffice_Level2)

[（三） 关于This](#_Toc3554_WPSOffice_Level2) [4](#_Toc3554_WPSOffice_Level2)

[（四） Finalize](#_Toc2181_WPSOffice_Level2) [4](#_Toc2181_WPSOffice_Level2)

[（五） 同步](#_Toc10460_WPSOffice_Level2) [5](#_Toc10460_WPSOffice_Level2)

**[第三章 String相关](#_Toc23143_WPSOffice_Level1)** **[5](#_Toc23143_WPSOffice_Level1)**

[1． 字符串拼接](#_Toc6597_WPSOffice_Level2) [5](#_Toc6597_WPSOffice_Level2)

[2． 待添加](#_Toc2354_WPSOffice_Level2) [6](#_Toc2354_WPSOffice_Level2)

[3． 注意事项](#_Toc3233_WPSOffice_Level2) [6](#_Toc3233_WPSOffice_Level2)

**[第四章 泛型](#_Toc13718_WPSOffice_Level1)** **[6](#_Toc13718_WPSOffice_Level1)**

[（一） 关于Class类](#_Toc1183_WPSOffice_Level2) [6](#_Toc1183_WPSOffice_Level2)

**[第五章 容器（Collection框架）](#_Toc28463_WPSOffice_Level1)** **[6](#_Toc28463_WPSOffice_Level1)**

[（一） 注意事项](#_Toc28533_WPSOffice_Level2) [7](#_Toc28533_WPSOffice_Level2)

[（二） List](#_Toc26996_WPSOffice_Level2) [7](#_Toc26996_WPSOffice_Level2)

[（三） Set](#_Toc29478_WPSOffice_Level2) [8](#_Toc29478_WPSOffice_Level2)

[（四） Map](#_Toc14703_WPSOffice_Level2) [8](#_Toc14703_WPSOffice_Level2)

**[第六章 多线程](#_Toc3554_WPSOffice_Level1)** **[9](#_Toc3554_WPSOffice_Level1)**

[（一） 线程锁](#_Toc7460_WPSOffice_Level2) [9](#_Toc7460_WPSOffice_Level2)

[（二） 线程复用](#_Toc27720_WPSOffice_Level2) [9](#_Toc27720_WPSOffice_Level2)

[（三） ThreadLocal](#_Toc30273_WPSOffice_Level2) [9](#_Toc30273_WPSOffice_Level2)

[（四） 并行与实现模式](#_Toc32665_WPSOffice_Level2) [10](#_Toc32665_WPSOffice_Level2)

**[第七章 多进程](#_Toc2181_WPSOffice_Level1)** **[10](#_Toc2181_WPSOffice_Level1)**

[（一） 通信方式](#_Toc1402_WPSOffice_Level2) [10](#_Toc1402_WPSOffice_Level2)

[（二） 共享内存](#_Toc1802_WPSOffice_Level2) [10](#_Toc1802_WPSOffice_Level2)

**[第八章 性能检测与优化](#_Toc10460_WPSOffice_Level1)** **[10](#_Toc10460_WPSOffice_Level1)**

[（三） 性能检测](#_Toc7664_WPSOffice_Level2) [10](#_Toc7664_WPSOffice_Level2)

[（四） 优化](#_Toc18158_WPSOffice_Level2) [10](#_Toc18158_WPSOffice_Level2)

[（五） 单元测试](#_Toc404_WPSOffice_Level2) [11](#_Toc404_WPSOffice_Level2)

**[第九章 IO](#_Toc6597_WPSOffice_Level1)** **[11](#_Toc6597_WPSOffice_Level1)**

[（一） File操作](#_Toc29873_WPSOffice_Level2) [11](#_Toc29873_WPSOffice_Level2)

[（二） Network IO](#_Toc9820_WPSOffice_Level2) [11](#_Toc9820_WPSOffice_Level2)

**[第十章 关于Java虚拟机](#_Toc2354_WPSOffice_Level1)** **[13](#_Toc2354_WPSOffice_Level1)**

[（一） 配置](#_Toc21446_WPSOffice_Level2) [13](#_Toc21446_WPSOffice_Level2)

[（二） 内存对象模型](#_Toc6649_WPSOffice_Level2) [13](#_Toc6649_WPSOffice_Level2)

[（三） 垃圾回收](#_Toc23287_WPSOffice_Level2) [14](#_Toc23287_WPSOffice_Level2)

**[第十一章 异常处理](#_Toc3233_WPSOffice_Level1)** **[14](#_Toc3233_WPSOffice_Level1)**

[1． Finally](#_Toc8296_WPSOffice_Level2) [14](#_Toc8296_WPSOffice_Level2)

[2． 常见异常](#_Toc11589_WPSOffice_Level2) [15](#_Toc11589_WPSOffice_Level2)

**[第十二章 源码解析](#_Toc1183_WPSOffice_Level1)** **[15](#_Toc1183_WPSOffice_Level1)**

[（一） Okhttp](#_Toc19408_WPSOffice_Level2) [15](#_Toc19408_WPSOffice_Level2)

[（二） Okio](#_Toc19131_WPSOffice_Level2) [15](#_Toc19131_WPSOffice_Level2)

[（三） 阿斯蒂芬](#_Toc138_WPSOffice_Level2) [15](#_Toc138_WPSOffice_Level2)

**[第十三章 其他](#_Toc28533_WPSOffice_Level1)** **[15](#_Toc28533_WPSOffice_Level1)**

[1． 随机值](#_Toc16605_WPSOffice_Level2) [15](#_Toc16605_WPSOffice_Level2)

[2． Volatile](#_Toc11856_WPSOffice_Level2) [15](#_Toc11856_WPSOffice_Level2)

**[第十四章 数据库](#_Toc26996_WPSOffice_Level1)** **[16](#_Toc26996_WPSOffice_Level1)**

[1． SQL优化](#_Toc8099_WPSOffice_Level2) [16](#_Toc8099_WPSOffice_Level2)

**[第十五章 常用库](#_Toc29478_WPSOffice_Level1)** **[16](#_Toc29478_WPSOffice_Level1)**

[1． Dubbo](#_Toc4058_WPSOffice_Level2) [16](#_Toc4058_WPSOffice_Level2)

[2． SpringMVC](#_Toc24340_WPSOffice_Level2) [16](#_Toc24340_WPSOffice_Level2)

**[第十六章 知识单](#_Toc14703_WPSOffice_Level1)** **[16](#_Toc14703_WPSOffice_Level1)**

[（一） 清单如下](#_Toc3191_WPSOffice_Level2) [16](#_Toc3191_WPSOffice_Level2)

# 设计模式

## 类的设计

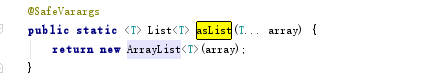
### 静态内部类。

静态内部类可以声明普通成员变量和方法，而普通内部类不能声明static成员变量和方法。

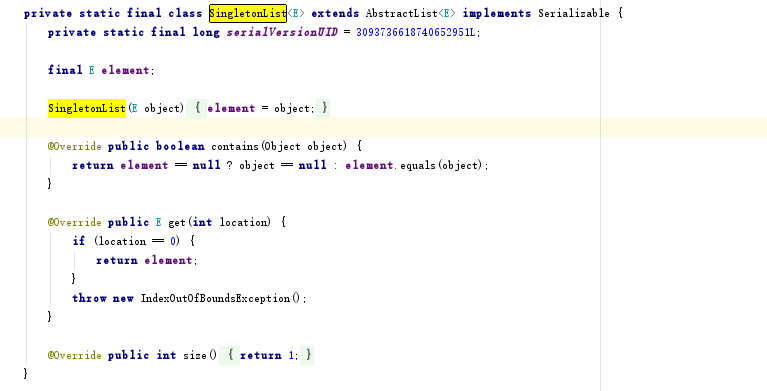
#### private静态内部类

范例代码：Arrays.asList()返回的就是有别于外部ArrayList的静态内部类，代码如下所示：





Collections的SingletonList。代码如下所示：



#### Build模式

如果类的构造器或静态工厂中有多个参数，设计这样类时，最好使用Builder模式，特别是当大多数参数都是可选的时候。如果现在不能确定参数的个数，最好一开始就使用构建器即Builder模式。OKHttp里面用到了Builder模式。

### 设计原则

泛型的继承：子类必须单独申明泛型类型，如果子类的泛型类型小于或不等于基类泛型，那么需要为基类指定具体类型。

## 解耦实现高效

### 啊手动阀

### 啊手动阀

# 关于Object

Object是所有类的基类。

## Class

对asdf

## Clone

## HashCode

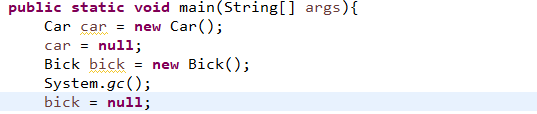
### Asd

## 关于This

This指针的指代是在何时决议的？

## Finalize

在实际应用中，不要依赖于使用finalize方法回收任何短缺的资源，这是因为很难知道这个方法什么时候才能够调用。Finalize执行的时机是，当GC解析出当前对象没有被引用的时候，GC会调用当前对象的Finalize方法。所以需要在子类中重载该方法以释放系统资源或执行其他清理操作。只要在执行System.gc()之前，堆对象还有GC ROOT（当前线程栈、静态区）在引用，此对象就不会执行finalize方法，例如如下代码，由于在执行gc之前线程栈中的变量bick引用了Bick()对象，所以此对象不会执行Finalize方法：



英文资料如下：Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. **Caution:** Never depend on finalize() for releasing limited resources such as file descriptors. For example, if an application object opens files, expecting that its finalize() method will close them, the application might find itself unable to open additional files when a tardy virtual machine is slow to call finalize(). What makes this problem worse is that finalize() might be called more frequently on another virtual machine, resulting in this too-many-open-files problem not revealing itself. [Note: the finalize() method has often been used to perform resurrection(make an unreferenced object referenced) to implement object pools that recycle the same objects when these objects are expensive (time-wise) to create (database connection objects are](https://books.google.co.jp/books?id=Up8QAwAAQBAJ&pg=PA155&lpg=PA155&dq=%22finalize%22++%26+%22android%22&source=bl&ots=asVyzKm57U&sig=t_hRvwIS_XZiFsUU0VmIlsn5NVc&hl=en&sa=X&redir_esc=y" \l "v=onepage&q=%22finalize%22%20%20%26%20%22android%22&f=false) .

此方法只执行一次，即使对象被复活，如果已经执行过了 finalize 方法，再次被 GC 时也不会再执行了，原因是： 含有 finalize 方法的 object 是在 new 的时候由虚拟机生成了一个 finalize reference 在来引用到该Object的，而在 finalize 方法执行的时候，该 object 所对应的 finalize Reference 会被释放掉，即使在这个时候把该 object 复活(即用强引用引用住该 object )，再第二次被 GC 的时候由于没有了 finalize reference 与之对应，所以 finalize 方法不会再执行。含有Finalize方法的object需要至少经过两轮GC才有可能被释放，为什么？（参考[链接](https://blog.csdn.net/a394268045/article/details/51735456)， 参考[链接](https://howtodoinjava.com/core-java/basics/why-not-to-use-finalize-method-in-java/)） ，如何判断一个对象有无回收可通过如下代码：

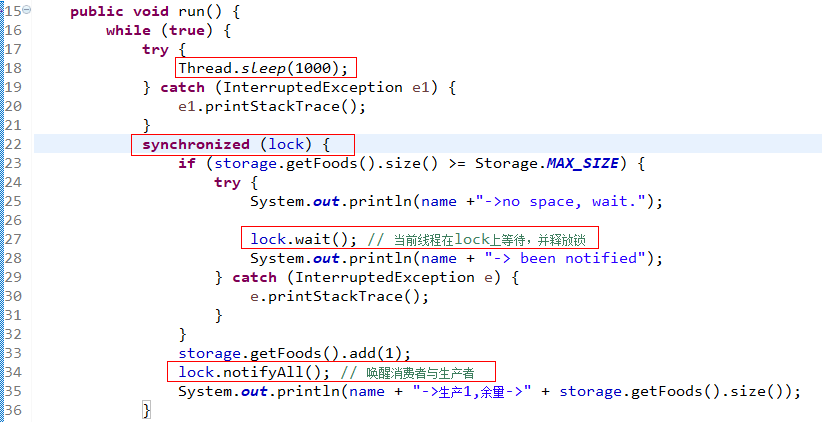
## 同步

对象Java对象是天生的Monitor。每一个Java对象都有成为Monitor的“潜质”。这是为什么？因为在Java的设计中，每一个对象自打娘胎里出来，就带了一把看不见的锁，通常我们叫“内部锁”，或者“Monitor锁”，或者“Intrinsic lock”。为了装逼起见，我们就叫它Intrinsic lock吧。有了这个锁的帮助，只要把类的所有对象方法都用synchronized关键字修饰，并且所有域都为私有（也就是只能通过方法访问对象状态），就是一个货真价实的Monitor了。（参考[链接](http://www.cnblogs.com/tomsheep/archive/2010/06/09/1754419.html)，Java.lang.ref.ReferenceQueue中用到了object.wait和object.notify，Android中的HandlerThread用到了wait和Notify）。

当前线程调用wait会使得当前线程阻塞，直到调用了notify。和获取同步锁阻塞是不同条件。在调用object.wait或objet.notify之前，必须先获得object的同步锁。wait和notify用在生产者和消费者模式中，即调用wait的线程是在等待其他线程调用notify才能继续往下执行，如下代码，如果当前现在在调用wait的时候阻塞了，会立即释放lock对象锁。notify()调用后，并不是马上就释放对象锁的，而是在相应的synchronized(){}语句块执行结束，自动释放锁后，JVM会在wait()对象锁的线程中随机选取一线程，赋予其对象锁，唤醒线程，继续执行（从代码运行上来看，被notify的线程会优先于其他线程获得CPU资源，即优先执行）。如果是notifyAll()就会释放所有的锁。

如下图所示，标号18代码行Thread.sleep(1000)休眠当前线程1s，即释放CPU资源。标号27处，执行lock.wait，释放锁，并且释放CPU资源，等待其他线程调用notify后，继续从标号28处开始执行。（关于wait和synchronize实现的内部关系，参见[链接](https://blog.csdn.net/boling_cavalry/article/details/77793224)）

为何ReferenceQueue和Android中的HandlerThread都在while中用wait？ 下面代码有个潜在错误，如果有2个线程在执行lock.wait后，都阻塞在标号27处，其他线程调用了lock.notify，两个线程都会继续从标号28处执行，导致逻辑错误，即总数会超过MAX\_SIZE。即标号23处代码要改为whie，只要大于MAX\_SIEZE就继续阻塞，避免了由if一次判断造成的逻辑不一致。放在while里面，是防止出于waiting的对象被别的原因调用了唤醒方法，但是while里面的条件并没有满足（也可能当时满足了，但是由于别的线程操作后，又不满足了），就需要再次调用wait将其挂起。



# String相关

String是。。。

### 字符串拼接

StringBuilder（没有线程同步）和StringBuffer(所有操作都是用synchronized对对象同步了，是线程安全的)。单线程下大量操作用StringBuilder，多线程下大量操作用StringBuffer，少量字符串操作用String。

### 待添加

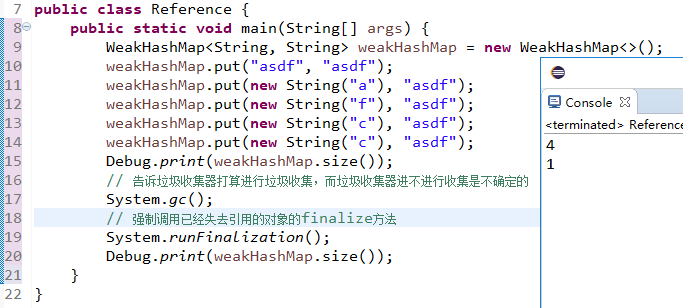
### 注意事项

#### SubString

代码String strA = strB.SubString(beginIndex , endIndex)，其中strA内部char[]指向strB所持有的char[]的一部分。要想strB和strA持有不同对象的引用，代码String strA = new String(strB.SubString(beginIndex, endIndex));

#### 关于字符串引用

代码如下：



在以上代码中，第13和14行，new String(“c”)作为key，执行了两次，由于equals结果相等，所以重复添加无效；第10行，由于“asdf”是在常量区内，不会被gc回收，所以执行强制回收内存之前map的数据是4个，回收之后是1个（即常量区的未回收）。

# 泛型

Java泛型是在JDK1.5之后引入的。把类型作为参数，让代码更加简洁。特点是：

1. 只在编译阶段有效，即可通过反射（执行在运行时）绕过泛型的类型检测。
2. 通过知道使用泛型定义的变量的类型限制，编译器可以更有效地提高Java程序的类型安全。
3. 消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。所有的强制转换都是自动和隐式的。
4. 泛型可以把使用Object的错误提前到编译后，而不是运行后，提升安全性。
5. 可以用<?>通配符。
6. 可以用<? super ABC>通配符。
7. 可以用<? extends ABC>通配符。

注意事项：

1. 泛型的类型参数只能是类类型（包括自定义类），不能是简单类型。
2. 泛型的类型参数可以有多个。
3. 没有泛型数组的说法。
4. 泛型参数类型声明必须在返回类型之前。

## 关于Class类

虚拟机通常使用运行时类型信息选准正确方法去执行，用来保存这些类型信息的类是Class类。Class类封装一个对象和接口运行时的状态，当装载类时，Class类型的对象自动创建。Class类的作用是运行时提供或获得某个对象的类型信息，和C++中的typeid()函数类似。这些信息也可用于反射。

# 容器（Collection框架）

实现Collection接口的主要是Set，List和Queue。Collections封装了很多静态方法，提供对Collection操作的实现，其中包括获取线程安全的容器（这些容器均以私有静态内部类实现，其中包括：SynchronizedMap, SynchronizedRandomAccessList, SynchronizedList, SynchronizedSortedSet, UnmodifiableMap等）。

内部使用数组实现的容器有：；内部使用红黑树实现的的容器有：；内部使用链表实现的容器有：；。Java提供的线程安全的容器都在util.concurrent里面，并以concurrent打头。各个容器特点如下表所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 名称 | 特点 | 适用场景 | 遍历 | 插入删除 | 随机访问 |
| List |  |  | 快 | 慢 | 快 |
| Map | 保存的是键值对；实现的是Map接口；存取比HashSet快； | 1、适合海量数据，o(1)的随机访问速度，不是可遍历； |  |  |  |
| Set |  | 1、不能有重复值的情形；2、 |  | 快 |  |
| Queue |  |  |  |  |  |

## 注意事项

* 频繁执行Collection.contain()，不能用list，可选用Map或者Set。（参考[链接](https://blog.csdn.net/fenglibing/article/details/9021201)。参考[链接](https://blog.csdn.net/fenglibing/article/details/8905007)。参考[链接](https://www.cnblogs.com/deepnighttwo/archive/2006/07/07/1964336.html)。）

同样都是循环遍历，为何arrayList的效率会相对低呢？因为hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的。\*此处需要阅读一下HashSet的contains源码，看看有何不同。\*

* 实际上，线程安全的容器，特别是Map，应用场景没有想象中的多，很多情况下一个业务会涉及容器的多个操作，即复合操作，并发执行时，线程安全的容器只能保证自身的数据不被破坏，但无法保证业务的行为是否正确。

## List

### ArrayList和LinkedList

当插入的数据量很小时，两者区别不太大，当插入的数据量大时，大约在容量的1/10之前，LinkedList会优于ArrayList，在其后就劣与ArrayList，且越靠近后面越差。所以个人觉得，一般首选用ArrayList，由于LinkedList可以实现栈、队列以及双端队列等数据结构，所以当特定需要时候，使用LinkedList，当然咯，数据量小的时候，两者差不多，视具体情况去选择使用；当数据量大的时候，如果只需要在靠前的部分插入或删除数据，那也可以选用LinkedList，反之选择ArrayList反而效率更高。

### 各种list的区别

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 描述 | 使用场景 | 优点 | 缺点 |
| CopyOnWriteArrayList | 1、不是利用锁机制进行线程安全，而是利用赋值新的进行替换进行； | 读操作明显多于写操作的情形。 |  | 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。 |
| ArrayList | [参考网址1](http://blog.csdn.net/qq_16318981/article/details/54632223). | 1、多数时候是按照索引访问元素；2、 |  |  |
| Vector | 1、线程安全的；2、 |  |  |  |
| LinkedList |  | 1、多数时候添加删除元素； |  |  |
| Arrays |  |  |  |  |
| Stack |  |  |  |  |

### 各种Queue的区别

队列就是有序的list，其中包括阻塞队列和非阻塞队列。阻塞队列BlockingQueue是接口，具有如下特征：1、阻塞的线程安全队列；2、插入数据时，如果已满，则线程等待直到可以插入为止；获取数据时，如果为空，则线程阻塞直到可以读取为止。阻塞队列使用的例子就是生产者消费者模式,也是各种实现生产者消费者模式方式中首选的方式。使用者不用关心什么阻塞生产，什么时候阻塞消费，使用非常方便。

|  |  |  |
| --- | --- | --- |
| 名称 | 描述 | 使用场景 |
| LinkedBlockingQueue | 1、阻塞队列；2、 |  |
| ArrayBlockingQueue | 1、阻塞队列；2、有界数组队列，内部是用数组实现，一旦设置初始大小，则无法更改； |  |
| SynchronousQueue | 1、阻塞队列；2、只能存放一个元素； |  |
| DelayQueue | 1、阻塞队列；2、对元素进行持有直到一个特定的延迟到期，注入其中的元素必须实现 java.util.concurrent.Delayed 接口； |  |
| PriorityBlockingQueue | 1、阻塞队列；2、无界的；3、无法插入null值；4、插入值必须实现java.lang.Comparable接口，队列中元素的排序就取决于这个 Comparable的 实现； |  |
| PriorityQueue |  |  |
| ConcurrentLinkedQueue |  |  |
| Deque |  |  |

## Set

### 各种set的区别。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 描述 | 使用场景 | 优点 | 缺点 |
| AbstractSet |  |  |  |  |
| HashSet | [参考链接1](http://blog.csdn.net/cloudeagle_bupt/article/details/73011694).[参考链接2](https://www.cnblogs.com/javabg/p/7258550.html). |  |  |  |
| TreeSet |  |  |  |  |
| SortedSet |  |  |  |  |
| SingletonSet |  |  |  |  |
| CopyOnWriteArraySet | 自Java1.5引入。 | 用于读多写少的并发场景 |  |  |
| EnumSet |  | EnumSet在内部以位向量的形式存储，这种存储形式非常紧凑、高效。因此EnumSet对象占用内存很小，而且运行效率很好。尤其是进行批量操作（如调用containsAll()和retainAll()方法）时，如果其参数也是EnumSet集合，则该批量操作的执行速度也非常快。 |  |  |

## Map

### 各种map的区别。

HashMap、HashTable及HashSet的区别，[参考网址](https://www.cnblogs.com/javabg/p/7258550.html)。

差异如下表所示：

|  |  |  |
| --- | --- | --- |
| **名称** | **描述** | **使用场景** |
| HashMap | [参考链接1](https://zhuanlan.zhihu.com/p/21673805)；1、允许使用一个Null作为key；2、非线程安全； |  |
| HashSet |  |  |
| HashTable | 1、线程安全的；2、键值未非Null；3、使用全局锁，导致性能低； |  |
| LinkedHashMap | 1、迭代访问效率高；2、进入的顺序与被取出的顺序一致； | 1、2、 |
| TreeMap | 1、红黑树实现；2、可排序； |  |
| WeakHashMap | 1、和hashmap不同点在于WeakHashMap的key只保留对实际对象的弱引用，这意味着当垃圾回收了该key所对应的实际对象后，WeakHashMap会自动删除该key对应的key-value对。 |  |
| EnumMap |  |  |
| IdentityHashMap | 1、和HashMap基本相似，在比较key的时候有不同（HashMap只要求key1.equals(key2)且hashcode相等；IdentityHashMap要求key1==key2）。 |  |
| ConcurrentHashMap | 1、线程安全的；2、对其中某一段加锁，互不影响。 |  |
| SortedMap |  |  |
| ConcurrentSkipListMap |  |  |
| SynchronizedMap |  |  |
| unmodifiableMap |  |  |

### 注意事项

* 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。
* 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。
* HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。
* JDK1.8引入红黑树大程度优化了HashMap的性能。
* 遍历map.keySet().iterator()效率低（实际上遍历了2次），map.entrySet().iterator()效率高。
* 如何判断map包含一个元素，通过contains或get() != null，还是遍历keyset或是遍历entrySet？通过实践，发现get() != null效率比较高，所用时间少了好几倍，get操作为什么快？（参考[链接](https://www.cnblogs.com/bailiyi/p/3730508.html)。）
* CurrentHashMap和HashMap的内存占用与get操作性能比较。（参考[链接](https://blog.csdn.net/ytbigdata/article/details/51861344)。）

# 多线程

[参考网址](https://mp.weixin.qq.com/s?__biz=MzA3NTYzODYzMg==&mid=2653578882&idx=2&sn=448e12f89436295162a5fa63bd623383&chksm=84b3b485b3c43d93fe5b794f3e0ffbb42704a3fef7d3a0675ee2594b2fb88180ed30695b6d7c&scene=21" \l "wechat_redirect)。线程的同步也是并发线程操作共享变量所带来的问题。多线程允许使用synchronize、volatile、ThreadLocal来保证多线程的安全。synchronize是一个重量级的锁，直接使得线程阻塞，单线程顺利执行，对于更新变量不会有并发操作，很大程度的降低的系统的性能。volatile是一个轻量级的原子锁，对于volatile修饰的变量，每一次的读和写，都必须和主内存交互，他禁止了编译器和处理器的一些重排序优化。 线程之间的通信是依靠共享内存和线程方法的调用来实现。在多线程的体系下，Java的内存模型分为主内存和共享内存，通过内存之间的数据交换，依赖多线程的可见性，实现线程之间的通信；线程具有基本状态，主动调用线程的wait、notify方法也可以实现线程之间的通信。

## 线程锁

Synchronized和lock区别：作为mutex的对象是不能变的，否则锁失效。

### 读写锁（ReadWriteLock）

JDK5中提供读写分离锁，即访问操作和写争夺临界区则上锁，读线程之间不上锁。

### 倒计时（CountDownLatch）

Asdfasd f

### 循环栅栏（ClyclicBarrier）

Asdfasd

### 锁的优化

啊手动阀撒旦

### 无锁的实现

### 死锁

## 线程复用

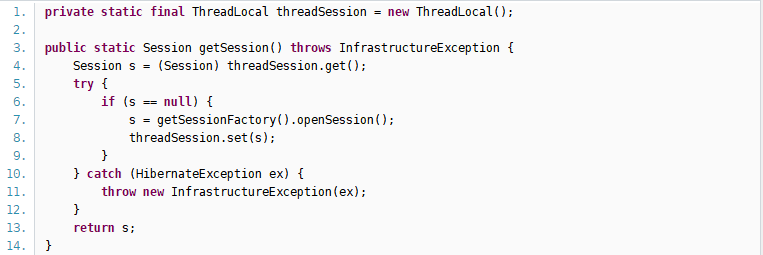
### 线程池

ThreadPoolExecutor类是线程池中最核心的一个类。[参考链接1](http://blog.csdn.net/wuseyukui/article/details/49617187).

## ThreadLocal

### 应用场景

最常见的ThreadLocal使用场景为 用来解决数据库连接、Session管理等。比如hibernate中典型的ThreadLocal的应用：



ThreadLocal不是用来解决对象共享访问问题的，而主要是提供了保持对象的方法和避免参数传递的方便的对象访问方式。归纳了两点：   
1、每个线程中都有一个自己的ThreadLocalMap类对象，可以将线程自己的对象保持到其中，各管各的，线程可以正确的访问到自己的对象。   
2、将一个共用的ThreadLocal静态实例作为key，将不同对象的引用保存到不同线程的ThreadLocalMap中，然后在线程执行的各处通过这个静态ThreadLocal实例的get()方法取得自己线程保存的那个对象，避免了将这个对象作为参数传递的麻烦。

### 实现原理

参考链接，[地址](https://www.cnblogs.com/dolphin0520/p/3920407.html)。.

## 并行与实现模式

### 待添加

### 待添加

# 多进程

## 通信方式

## 共享内存

# 性能检测与优化

## 性能检测

性能包括：等待时间、吞吐量、利用率、效率、容量、扩展性、退化。

## 优化

### 内存优化与缓存

#### 缓存

缓存的母的是加快响应速度、缓解数据库查询压力，但存在缓存穿透和缓存雪崩的问题。

## 单元测试

# IO

## File操作

|  |  |  |  |
| --- | --- | --- | --- |
| **概念名称** | **特征** | **描述** | **注意事项** |
| ByteBuffer |  |  |  |
| BufferedReader |  |  |  |
| BufferedInputStream |  |  |  |
| MappedByteBuffer | 1. flip()把指针调整到开始位置。 2. Put() | 1、将文件映射为内存文件；2、实现多进程之间共享内存；3、**效率要比read和write系统调用高**（read()是系统调用，首先将文件从硬盘拷贝到内核空间的一个缓冲区，再将这些数据拷贝到用户空间，实际上进行了两次数据拷贝；map()也是系统调用，但没有进行数据拷贝，当缺页中断发生时，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝。） | 1、如果使用了FileChannel.map方法去映射一个文件，然后马上关闭这个channel，然后再试图删除文件，就会发现不能成功。这是因为MappedByteBuffer还没有被回收，文件句柄还没有释放。而具体什么时候才会释放，以及能不能提前释放。2、MappedByteBuffer在处理大文件时的确性能很高，但也存在一些问题，如内存占用、文件关闭不确定，被其打开的文件只有在垃圾回收的才会被关闭，而且这个时间点是不确定的。3、在数据量很小的时候，因为direct buffer的初始化时间较长，所以只有在数据量较大的时候用。 |
| FileChannel |  | 1、用于在文件的输入和输出之间建立通道，提高了传输效率。2、提供了之间将文件映射到内存的方法。3、fileChannelFrom.transferTo(0, fileChannelFrom.size(), fileChannelTo); | 1、阻塞模式；2、只能通过通过InputStream，OutputStream或RandomAccessFile获取。3、force()方法将所有未写入的数据从通道刷新到磁盘中。 |
| FileInputStream |  |  |  |
| FileOutputStream |  |  |  |
| BufferedInputStream（[链接1](http://blog.csdn.net/fcbayernmunchen/article/details/8635427)，链接2） |  |  |  |
| 堆外内存DirectByteBuffer  （[连接1](https://www.jianshu.com/p/007052ee3773)， [连接2](http://lovestblog.cn/blog/2015/05/12/direct-buffer/)，） |  |  |  |
| FileLock |  |  |  |
| RandomAccessFile |  |  |  |
| DataInputStream |  |  |  |
| DataOutputStream |  |  |  |

## Network IO

### 各种网络库对比

|  |  |  |  |
| --- | --- | --- | --- |
| **概念名称** | **特征** | **方法描述** | **注意事项** |
| Zerocopy(Netty)（[链接1](https://www.ibm.com/developerworks/library/j-zerocopy/)，[连接2](http://xcorpion.tech/2016/09/10/It-s-all-about-buffers-zero-copy-mmap-and-Java-NIO/)，[连接3](http://www.cnblogs.com/hapjin/p/5736188.html)） |  | 专用于处理Socket，核心概念是channel和eventloop前者抽象了io模型如tcp http 等。 | Netty并发高、传输快、封装好；更好的吞吐量、低延迟；（有了Netty，你可以实现自己的HTTP服务器，FTP服务器，UDP服务器，RPC服务器，WebSocket服务器，Redis的Proxy服务器，MySQL的Proxy服务器等等。） |
| dubbo |  |  |  |
|  |  |  |  |

### Netty

### OKhttp

### 通信安全

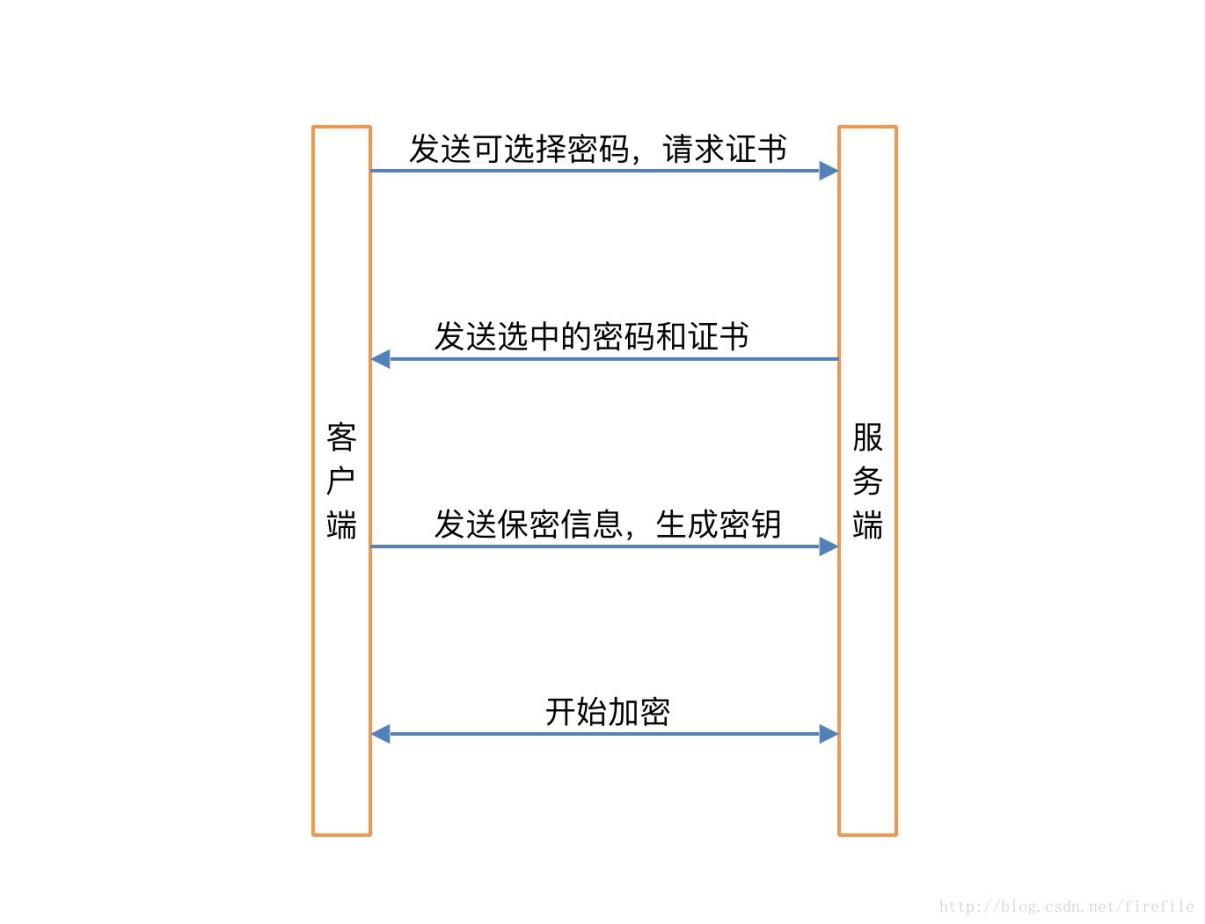
TLS 是进行 HTTPS 连接的重要环节，通过了 TLS 层进行协商，后续的 HTTP 请求就可以使用协商好的对称密钥进行加密。SSL 是 Netscape 开发的专门用来保护 Web 通讯，目前版本为 3.0。TLS 是 IETF 制定的新协议，建立在 SSL 3.0 之上。所以 TLS 1.0 可以认为是 SSL 3.1。TLS（Transport Layer Security Protocol） 协议分为两部分：TLS 记录协议；TLS 握手协议。

HTTPS 是在 HTTP 和 TCP 之间加了一层 TLS，这个 TLS 协商了一个对称密钥来进行 HTTP 加密。SSL/TLS 不仅仅可以用在 HTTP，也可以用在 FTP，Telnet 等应用层协议上。

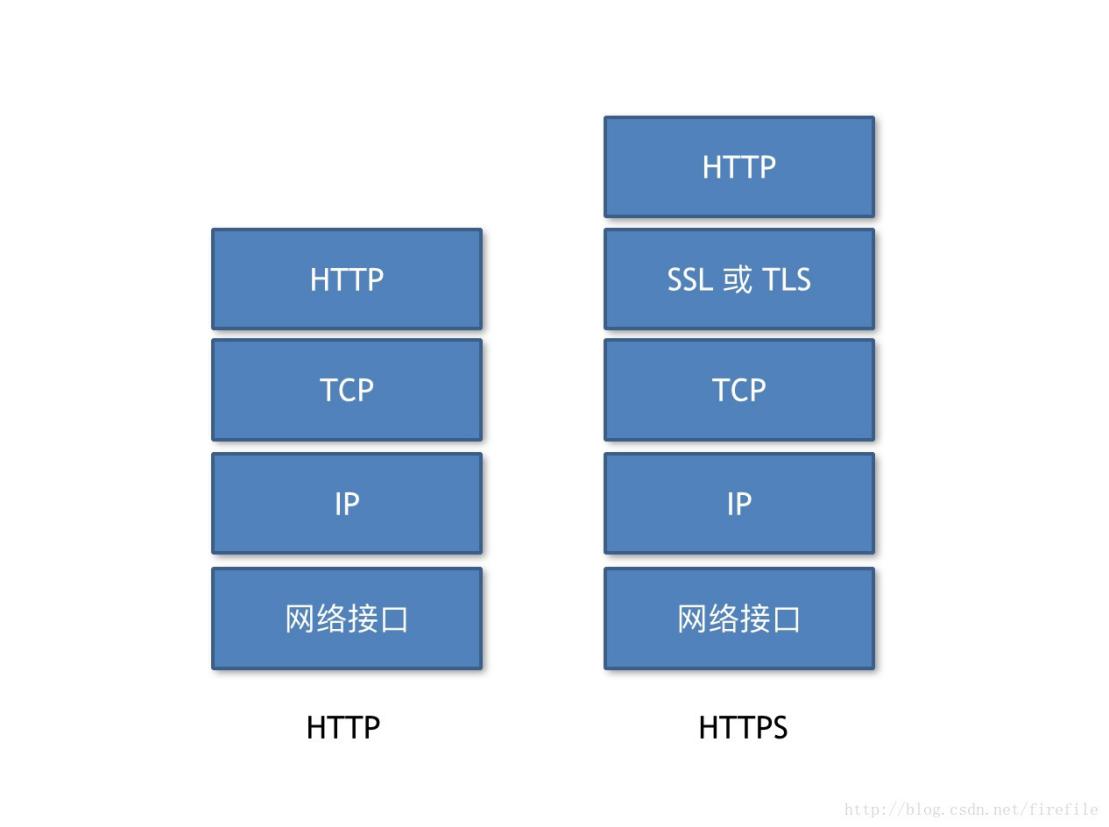
SL/TLS 实际上混合使用了对称和非对称密钥，主要分成这几步：

1. 使用非对称密钥建立安全的通道 ：
   1. 客户端请求 Https 连接，发送可用的 TLS 版本和可用的密码套件
   2. 服务端返回证书，密码套件和 TLS 版本
2. 用安全的通道产生并发送临时的随机对称密钥
   1. 生成随机对称密钥，使用证书中的服务端公钥加密，发送给服务端
   2. 服务端使用私钥解密获取对称密钥
   3. 使用对称密钥加密信息，进行交互

简单过程如下：



TLS原理如下图所示：

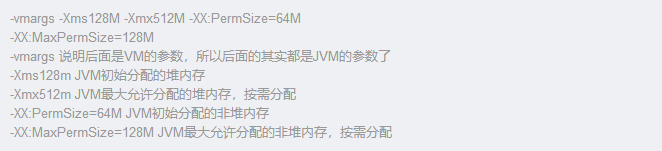


阿斯蒂芬

# 关于Java虚拟机

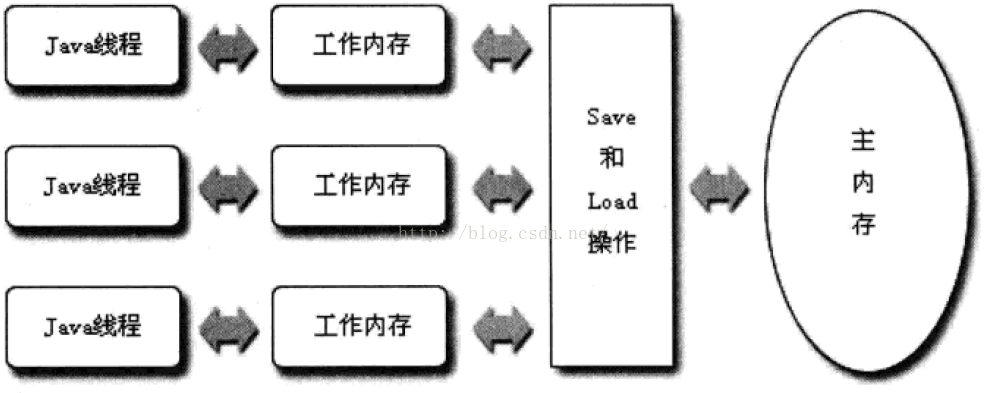
## 配置

Java虚拟机配置包括初始分配堆内存大小XMS、最大允许分配的对内存大小（按需分配）XMX、非堆内存大小PERMSIZE、最大允许的非堆内存大小（按需分配）MAXPERMSIZE，-Xms 为jvm启动时分配的内存，比如-Xms200m，表示分配200M，-Xmx 为jvm运行过程中分配的最大内存，比如-Xms500m，表示jvm进程最多只能够占用500M内存，-Xss 为jvm启动的每个线程分配的内存大小，默认JDK1.4中是256K，JDK1.5+中是1M。如下所示：



## 内存对象模型

Java 内存模型来屏蔽掉各种硬件和操作系统的内存差异，达到跨平台的内存访问效果。JLS(Java语言规范)定义了一个统一的内存管理模型JMM(Java Memory Model)。Java内存模型规定了所有的变量都存储在主内存中，此处的主内存仅仅是虚拟机内存的一部分，而虚拟机内存也仅仅是计算机物理内存的一部分（为虚拟机进程分配的那一部分）。Java内存模型分为主内存，和工作内存。主内存是所有的线程所共享的，工作内存是每个线程自己有一个，不是共享的。每条线程还有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝。线程对变量的所有操作（读取、赋值），都必须在工作内存中进行，而不能直接读写主内存中的变量。主内存和工作内存关系如下图所示（工作内存一定是栈内存吗？主内存一定是堆内存吗？）：



### 堆的概念

Java 虚拟机只有一个堆，即运行时的数据区域，所有类实例和数组的内存均从此处分配。堆是在Java 虚拟机启动时创建的。在JVM中堆之外的内存称为非堆内存(Non-heap memory)。可以看出JVM主要管理两种类型的内存：堆和非堆。

简单来说，堆就是Java代码可及的内存，是留给开发人员使用的，而非堆就是JVM留给自己用的，所以方法区、JVM内部处理或优化所需的内存(如JIT编译后的代码缓存)、每个类结构(如运行时常数池、字段和方法数据)以及方法和构造方法的代码都在非堆内存中。

### 堆的分配

JVM初始分配的堆内存由-Xms指定，默认是物理内存的1/64；JVM最大分配的堆内存由-Xmx指定，默认是物理内存的1/4。默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制；空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制。因此服务器一般设置-Xms、-Xmx 相等以避免在每次GC 后调整堆的大小。 疑问：Java虚拟机调整堆的大小有何影响？  
 说明：如果-Xmx不指定或者指定偏小，应用可能会导致java.lang.OutOfMemory错误，此错误来自JVM，不是Throwable的，无法用try…catch捕捉。

JVM使用-XX:PermSize设置非堆内存初始值，默认是物理内存的1/64；由XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的1/4。（还有一说：MaxPermSize缺省值和-server -client选项相关， -server选项下默认MaxPermSize为64m，-client选项下默认MaxPermSize为32m。这个我没有实验）上面错误信息中的PermGen space的全称是Permanent Generation space，是指内存的永久保存区域。还没有弄明白PermGen space是属于非堆内存，还是就是非堆内存，但至少是属于了。XX:MaxPermSize设置过小会导致java.lang.OutOfMemoryError: PermGen space 就是内存益出。疑问：如果操作系统虚拟内存大小小于物理内存大小，对性能有啥影响？例如32位操作系统允许在16G内存的物理机上。参考[网址](https://blog.csdn.net/qq_27258799/article/details/51599093)。[参考网址](https://mp.weixin.qq.com/s?__biz=MzIwNTc4NTEwOQ==&mid=2247485493&idx=1&sn=ee0b015dad691ded7f015ebbe684a275&chksm=972ad94fa05d50597768f1d4edf956e1308d80696b555eff42b63bc5984dd8ff4b6852798945&mpshare=1&scene=23&srcid=0614Z08gTHzjXH2TTNRGU1a9" \l "rd)。[参考网址](https://mp.weixin.qq.com/s?__biz=MzIwNTc4NTEwOQ==&mid=2247485284&idx=1&sn=12cd809f1f546562ae4a3643670b64c4&chksm=972ad61ea05d5f08f93ef7b812cf3cf3f6683b02ac212c26e00cc4b51a5d5f94f882cd656d41&mpshare=1&scene=23&srcid=0509MUi0DpTiwXKHBs92Ce6J&lan=cn&lan=cn&la)。

### 堆内存引用

1. 一般情况下，函数调用不能直接返回对堆内存的引用，而应该返回其内存的深度拷贝，返回堆原始堆内存引用会增加代码逻辑的复杂性，容易出现逻辑错误、多线程冲突等隐藏问题。
2. 引用类型分为：强引用（用户自行回收）、软引用（内存不足的时候回收）、弱引用（一旦开始垃圾回收即回收，不一定内存不足）。[参考网址](https://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247486083&idx=1&sn=11fb39b27fefb7ae1c8ce1a7bc077c3c&chksm=e9c5f332deb27a24182206a4a68710ee10ee1b7b053e262c416d52ff963b4193ddca0ef9a6f4&mpshare=1&scene=23&srcid=0517Scn7sprMtyrpDGtFOd9s" \l "rd)。
3. 我们希望继续对某个对象持有应用，并且希望gc能够回收对象，那么就可以考虑使用 Reference 对象。垃圾回收器能够根据不同类型的 Reference 派生类来决定是否回收 Reference 内包装的对象。必须保证只能通过 Reference 获得该对象，其他任何普通引用都没有指向该对象。如下图10.2.3.3所示，当GC检测到Reference对象中包含的key除了自身对它持有引用外，没有其他普通引用执行key对象，GC就会回收key对象。如果设定了ReferenceQueue队列，在key对象回收后，会将reference对象添加到ReferenceQueue队列中，但是reference不再持有key对象。
4. ReferenceQueue是作为 JVM GC与上层Reference对象管理之间的一个消息传递方式，它使得我们可以对所监听的对象引用可达发生变化时做一些处理。

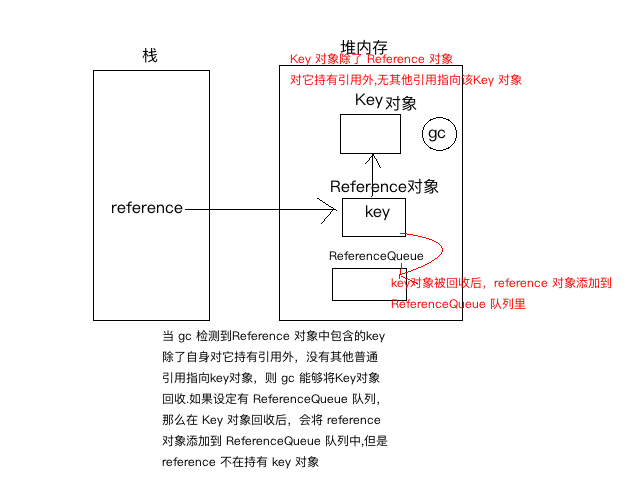


图 10.2.3.3

### 类成员生命周期

类成员包括静态的、非静态的、常量的、非常量的；同样包括内部类、函数、对象。下表描述了各种类型的生命周期：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 类型 | 赋值时机 | 加载时机 | 存在时间段 | 销毁时机 |
| 变量 | 静态 |  |  |  |  |
| 非静态 |  |  |  |  |
| 常量 |  |  |  |  |
| 非常量 |  |  |  |  |
| 类 | 非静态内部类 |  |  |  |  |

## 垃圾回收

啊手动阀（参考[链接](https://blog.csdn.net/a394268045/article/details/51735456)）。

# 异常处理

### Finally

* 不管有没有出现异常，finally块中代码都会执行；
* 当try和catch中有return时，finally仍然会执行；
* finally***是在return语句执行之后***，返回之前执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，不管finally中的代码怎么样，返回的值都不会改变，仍然是之前保存的值），所以***函数返回值是在finally执行前就已经确定了***；
* finally中***如果包含return，那么程序将在这里返回***，而不是try或catch中的return返回，***返回值就不是try或catch中保存的返回值了***。

### 常见异常

因为list可用保存null值，所以即便list有元素，也需要判空，防止NPE。

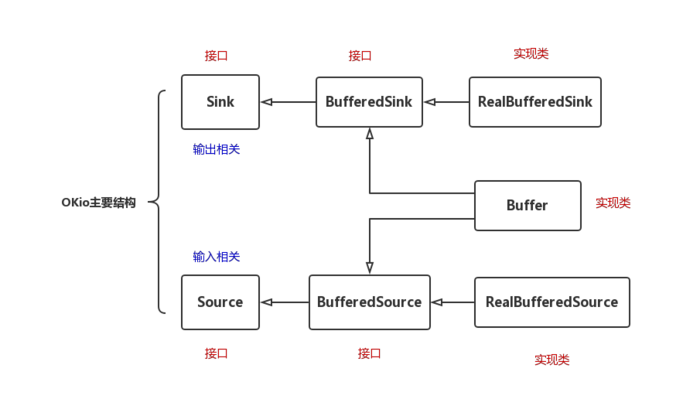
# 源码解析

## Okhttp

### 类的设计

## Okio

### 类的设计



### 详细

参考网址：[地址1](https://www.jianshu.com/p/58284634e4f8?utm_campaign=maleskine&utm_content=note&utm_medium=seo_notes&utm_source=recommendation).

## 阿斯蒂芬

# 其他

### 随机值

注意 Math.random() 这个方法返回是 double 类型，注意取值的范围 0≤x<1（能够取到零值，注意除零异常），如果想获取整数类型的随机数，不要将 x 放大 10 的若干倍然后取整，直接使用 Random 对象的 nextInt 或者 nextLong 方法。

### Volatile

Java内存模型将没有被volatile修饰的64位的数据的读写操作划分为两次32为的操作来进行，这样的话，多线程并发，就会存在线程可能读取到“半个变量”的值，不过，这种情况非常罕见，目前各平台的商用虚拟机几乎都选择把64位的读写作为原子操作来实现规范的。Java语言规范中指出：为了获得最佳速度，允许线程保存共享成员变量的私有拷贝，而且只当线程进入或者离开同步代码块时才与共享成员变量的原始值对比。这样当多个线程同时与某个对象交互时，就必须要注意到要让线程及时的得到共享成员变量的变化。而volatile关键字就是提示VM：对于这个成员变量不能保存它的私有拷贝，而应直接与共享成员变量交互。

使用建议：在两个或者更多的线程访问的成员变量上使用volatile。当要访问的变量已在synchronized代码块中，或者为常量时，不必使用。由于使用volatile屏蔽掉了VM中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

# 数据库

### SQL优化

[参考网址](https://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247486006&idx=1&sn=b4370640c163edcc91a8bb86fdb4cfa9&chksm=e9c5f387deb27a91756787d248de8d888aa8afa849cbbddd091c60e5adc08e1665d24f0fd517&mpshare=1&scene=23&srcid=0517Pe8jH10Qj1MOzb6Ykgbk" \l "rd)。

# 常用库

### Dubbo

### SpringMVC

# 知识单

靠技术吃饭，要拿高薪，那么试问自己，对原理了解多深，看过多少源码？

需要学习的知识点包括：中间件，例如消息队列（ActiveMQ, RabbitMQ, RocketMQ, Kafka, ZeroMQ实现解耦、异步及消峰），分布式（Dubbo、Zookeeper等负载均衡）几种算法的java实现。[参考链接1](https://zhuanlan.zhihu.com/p/33774158)。常见java[面试题目](https://mp.weixin.qq.com/s?__biz=MzI3ODcxMzQzMw==&mid=2247485944&idx=1&sn=483dd90ceab2e620879989e09d385b81&chksm=eb538ccedc2405d8d7c09d2bbdfe589539b83af18168f307ebbc32be980af378f113d200cbe5&mpshare=1&scene=23&srcid=0514F7PhPs0hVASHsiiVEtIb" \l "rd)。

## 清单如下

