Java基础

面向对象、四个特性、重载重写、static和final等等很多东西

面向对象

对象、类、**继承、封装、多态**

重载 & 重写

重载（Overloading）

方法重载是让类以统一的方式处理不同数据类型的手段。
一个类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法。
返回值类型可以相同也可以不相同，无法以返回型别作为重载函数的区分标准。

重写（Overriding）

子类对父类的方法进行重新编写。如果在子类中的方法与其父类有相同的的方法名、返回类型和参数表，我们说该方法被重写 (Overriding)。
如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。
子类函数的访问修饰权限不能低于父类的。

四个特性

1. 抽象
2. 继承
3. 封装
4. 多态

Static

static方法就是没有this的方法。在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法。这实际上正是static方法的主要用途。

> 方便在没有创建对象的情况下来进行调用（方法/变量）。

1. static方法

   在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用。虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。

   另外记住，即使没有显示地声明为static，类的构造器实际上也是静态方法。

2. static变量

static变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。

```
static成员变量的初始化顺序按照定义的顺序进行初始化。
```

3. static代码块

   static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。

- Java中的static关键字不会影响到变量或者方法的作用域。在Java中能够影响到访问权限的只有private、public、protected（包括包访问权限）这几个关键字。
- 静态成员变量虽然独立于对象，但是不代表不可以通过对象去访问，所有的静态方法和静态变量都可以通过对象访问（只要访问权限足够）。
- 在执行main方法之前，必须先加载所有的类（有父类的先加载），有static变量和static块的时候要执行。然后开始执行main函数。

final

final变量

凡是对成员变量或者本地变量(在方法中的或者代码块中的变量称为本地变量)声明为final的都叫作final变量。final变量经常和static关键字一起使用，作为常量。

```
public static final String LOAN = “loan”;
LOAN = new String(“loan”) //invalid compilation error
```

final方法

final也可以声明方法。方法前面加上final关键字，代表这个方法不可以被子类的方法重写。
final方法比非final方法要快，因为在编译的时候已经静态绑定了，不需要在运行时再动态绑定。

final类

使用final来修饰的类叫作final类。final类通常功能是完整的，它们不能被继承。Java中有许多类是final的，譬如String, Interger以及其他包装类。

好处

- final关键字提高了性能。JVM和Java应用都会缓存final变量。
- final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。
- 使用final关键字，JVM会对方法、变量及类进行优化。

集合

```
HashMap、HashTable、ConcurrentHashMap、各种List，最好结合源码看
```

HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。

- HashMap几乎可以等价于Hashtable，除了HashMap是非synchronized的，并可以接受null(HashMap可以接受为null的键值(key)和值(value)，而Hashtable则不行)。
- 这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。
- 由于Hashtable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢。如果你不需要同步，只需要单一线程，那么使用HashMap性能要好过Hashtable。
- HashMap不能保证随着时间的推移Map中的元素次序是不变的。

hashmap	线程不安全	允许有null的键和值	效率高一点、	方法不是Synchronize的要提供外同步	有containsvalue和containsKey方法	HashMap 是Java1.2 引进的 Map interface 的一个实现	HashMap是Hashtable的轻量级实现
hashtable	线程安全	不允许有null的键和值	效率稍低、	方法是是Synchronize的	有contains方法方法	、Hashtable 继承于 Dictionary 类	Hashtable 比HashMap要旧

参考资料：http://www.cnblogs.com/beatIteWeNerverGiveUp/p/5709841.html

> 面试经典！！！！

HashMap的工作原理是近年来常见的Java面试题。几乎每个Java程序员都知道HashMap，都知道哪里要用HashMap，知道Hashtable和HashMap之间的区别，那么为何这道面试题如此特殊呢？是因为这道题考察的深度很深。这题经常出现在高级或中高级面试中。投资银行更喜欢问这个问题，甚至会要求你实现HashMap来考察你的编程能力。ConcurrentHashMap和其它同步集合的引入让这道题变得更加复杂。让我们开始探索的旅程吧！

“你用过HashMap吗？”“什么是HashMap？你为什么用到它？”

几乎每个人都会回答“是的”，然后回答HashMap的一些特性，譬如HashMap可以接受null键值和值，而Hashtable则不能；HashMap是非synchronized;HashMap很快；以及HashMap储存的是键值对等等。这显示出你已经用过HashMap，而且对它相当的熟悉。但是面试官来个急转直下，从此刻开始问出一些刁钻的问题，关于HashMap的更多基础的细节。面试官可能会问出下面的问题：

“你知道HashMap的工作原理吗？”“你知道HashMap的get()方法的工作原理吗？”

你也许会回答“我没有详查标准的Java API，你可以看看Java源代码或者Open JDK。”“我可以用Google找到答案。”

但一些面试者可能可以给出答案，“HashMap是基于hashing的原理，我们使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。当我们给put()方法传递键和值时，我们先对键调用hashCode()方法，返回的hashCode用于找到bucket位置来储存Entry对象。”这里关键点在于指出，HashMap是在bucket中储存键对象和值对象，作为Map.Entry。这一点有助于理解获取对象的逻辑。如果你没有意识到这一点，或者错误的认为仅仅只在bucket中存储值的话，你将不会回答如何从HashMap中获取对象的逻辑。这个答案相当的正确，也显示出面试者确实知道hashing以及HashMap的工作原理。但是这仅仅是故事的开始，当面试官加入一些Java程序员每天要碰到的实际场景的时候，错误的答案频现。下个问题可能是关于HashMap中的碰撞探测(collision detection)以及碰撞的解决方法：

“当两个对象的hashcode相同会发生什么？”从这里开始，真正的困惑开始了，一些面试者会回答因为hashcode相同，所以两个对象是相等的，HashMap将会抛出异常，或者不会存储它们。然后面试官可能会提醒他们有equals()和hashCode()两个方法，并告诉他们两个对象就算hashcode相同，但是它们可能并不相等。一些面试者可能就此放弃，而另外一些还能继续挺进，他们回答“因为hashcode相同，所以它们的bucket位置相同，‘碰撞’会发生。因为HashMap使用链表存储对象，这

个Entry(包含有键值对的Map.Entry对象)会存储在链表中。”这个答案非常的合理，虽然有很多种处理碰撞的方法，这种方法是最简单的，也正是HashMap的处理方法。但故事还没有完结，面试官会继续问：

“如果两个键的hashcode相同，你如何获取值对象？” 面试者会回答：当我们调用get()方法，HashMap会使用键对象的hashcode找到bucket位置，然后获取值对象。面试官提醒他如果有两个值对象储存在同一个bucket，他给出答案:将会遍历链表直到找到值对象。面试官会问因为你并没有值对象去比较，你是如何确定确定找到值对象的？除非面试者直到HashMap在链表中存储的是键值对，否则他们不可能回答出这一题。

其中一些记得这个重要知识点的面试者会说，找到bucket位置之后，会调用keys.equals()方法去找到链表中正确的节点，最终找到要找的值对象。完美的答案！

许多情况下，面试者会在这个环节中出错，因为他们混淆了hashCode()和equals()方法。因为在此之前hashCode()屡屡出现，而equals()方法仅仅在获取值对象的时候才出现。一些优秀的开发者会指出使用不可变的、声明作final的对象，并且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

如果你认为到这里已经完结了，那么听到下面这个问题的时候，你会大吃一惊。“如果HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？”除非你真正知道HashMap的工作原理，否则你将回答不出这道题。默认的负载因子大小为0.75，也就是说，当一个map填满了75%的bucket时候，和其它集合类(如ArrayList等)一样，将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中。这个过程叫作rehashing，因为它调用hash方法找到新的bucket位置。

如果你能够回答这道问题，下面的问题来了：“你了解重新调整HashMap大小存在什么问题吗？”你可能回答不上来，这时面试官会提醒你当多线程的情况下，可能产生条件竞争(race condition)。

当重新调整HashMap大小的时候，确实存在条件竞争，因为如果两个线程都发现HashMap需要重新调整大小了，它们会同时试着调整大小。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。这个时候，你可以质问面试官，为什么这么奇怪，要在多线程的环境下使用HashMap呢？：）

热心的读者贡献了更多的关于HashMap的问题：

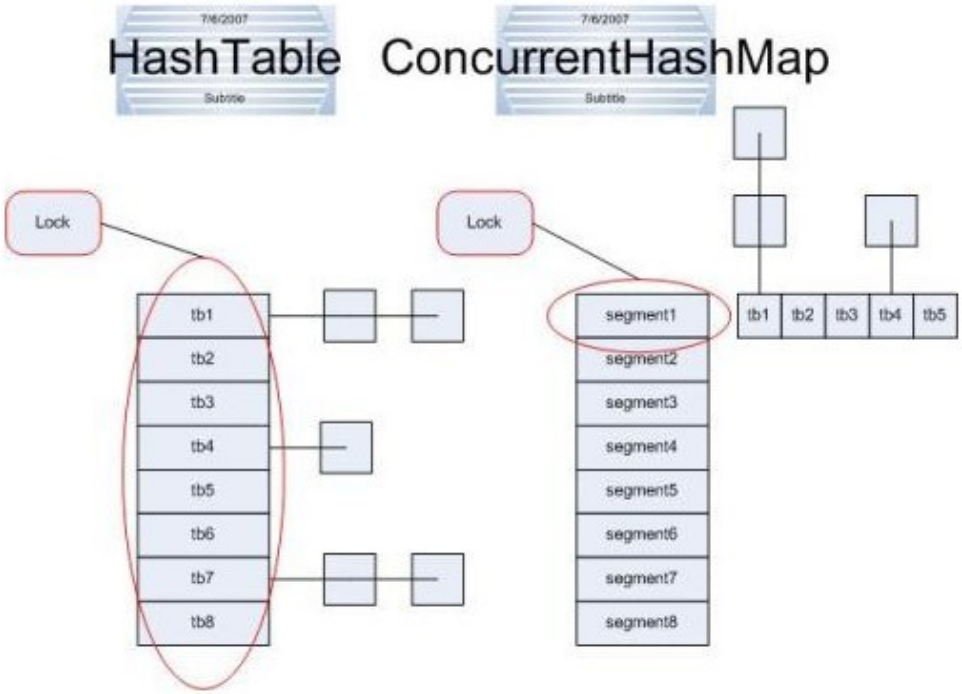
为什么String, Interger这样的wrapper类适合作为键？ String, Interger这样的wrapper类作为HashMap的键是再适合不过了，而且String最为常用。因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的hashcode的话，那么就不能从HashMap中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个field声明成final就能保证hashCode是不变的，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的hashcode的话，那么碰撞的几率就会小些，这样就能提高HashMap的性能。
我们可以使用自定义的对象作为键吗？ 这是前一个问题的延伸。当然你可能使用任何对象作为键，只要它遵守了equals()和hashCode()方法的定义规则，并且当对象插入到Map中之后将不会再改变了。如果这个自定义对象时不可变的，那么它已经满足了作为键的条件，因为当它创建之后就已经不能改变了。
我们可以使用CocurrentHashMap来代替Hashtable吗？这是另外一个很热门的面试题，因为ConcurrentHashMap越来越多人用了。我们知道Hashtable是synchronized的，但是ConcurrentHashMap同步性能更好，因为它仅仅根据同步级别对map的一部分进行上锁。ConcurrentHashMap当然可以代替HashTable，但是HashTable提供更强的线程安全性。

Hashtable和ConcurrentHashMap的区别

hashtable是做了同步的，hashmap未考虑同步。所以hashmap在单线程情况下效率较高。hashtable在的多线程情况下，同

步操作能保证程序执行的正确性。

但是hashtable每次同步执行的时候都要锁住整个结构。看下图：



图左侧清晰的标注出来，lock每次都要锁住整个结构。

ConcurrentHashMap正是为了解决这个问题而诞生的。

ConcurrentHashMap锁的方式是稍微细粒度的。 ConcurrentHashMap将hash表分为16个桶（默认值），诸如get,put,remove等常用操作只锁当前需要用到的桶。（Lock()函数）

试想，原来 只能一个线程进入，现在却能同时16个写线程进入（写线程才需要锁定，而读线程几乎不受限制，之后会提到），并发性的提升是显而易见的。

更令人惊讶的是ConcurrentHashMap的读取并发，因为在读取的大多数时候都没有用到锁定，所以读取操作几乎是完全的并发操作，而写操作锁定的粒度又非常细，比起之前又更加快速（这一点在桶更多时表现得更明显些）。只有在求size等操作时才需要锁定整个表。

```
参考资料：http://www.cnblogs.com/wang-meng/p/5808006.html
```

并发和多线程

```
线程池、SYNC和Lock锁机制、线程通信、volatile、ThreadLocal、
CyclicBarrier、Atom包、CountDownLatch、AQS、CAS原理等等
```

线程池

有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务?
在Java中可以通过线程池来达到这样的效果。

一.Java中的ThreadPoolExecutor类

二.深入剖析线程池实现原理

三.使用示例

四.如何合理配置线程池的大小

Java中的ThreadPoolExecutor类

java.uitl.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类。
ThreadPoolExecutor继承了AbstractExecutorService类.
AbstractExecutorService是一个抽象类，它实现了ExecutorService接口。
而ExecutorService又是继承了Executor接口

```
public ThreadPoolExecutor(
int corePoolSize,
int maximumPoolSize,
long keepAliveTime,
TimeUnit unit, B
lockingQueue<Runnable> workQueue,
ThreadFactory threadFactory,
RejectedExecutionHandler handler
);
```

1. **corePoolSize**：核心池的大小, 在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；
2. **maximumPoolSize**：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
3. **keepAliveTime**：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了**allowCoreThreadTimeOut(boolean)**方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；
4. **unit**：参数keepAliveTime的时间单位。
5. **workQueue**：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

   ```
   ArrayBlockingQueue;
   LinkedBlockingQueue;
   SynchronousQueue;
   ```

6. **threadFactory**：线程工厂，主要用来创建线程；

7. **handler**：表示当拒绝处理任务时的策略，有以下四种取值：

   ```
   ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
   ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
   ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务      （重复此过程）
   ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务
   ```

在ThreadPoolExecutor类中有几个非常重要的方法：

- **execute()**：Executor中声明的方法，在ThreadPoolExecutor进行了具体的实现，通过这个方法可以向线程池提交一个任务，交由线程池去执行。
- **submit()**：在ExecutorService中声明的方法，在AbstractExecutorService就已经有了具体的实现，在ThreadPoolExecutor中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和execute()方法不同，它能够返回任务执行的结果，去看submit()方法的实现，会发现它实际上还是调用的execute()方法，只不过它利用了Future来获取任务执行结果。
- **shutdown()和shutdownNow()**：用来关闭线程池的。

深入剖析线程池实现原理

1 线程池状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：

```
volatile int runState;
static final int RUNNING    = 0;
static final int SHUTDOWN   = 1;
static final int STOP       = 2;
static final int TERMINATED = 3;
```

runState表示当前线程池的状态，它是一个volatile变量用来保证线程之间的**可见性**；

当创建线程池后，初始时，线程池处于RUNNING状态；

如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务；

当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态。

2 任务的执行

在了解将任务提交给线程池到任务执行完毕整个过程之前，我们先来看一下ThreadPoolExecutor类中其他的一些比较重要成员变量：

```
private final BlockingQueue<Runnable> workQueue;
//任务缓存队列，用来存放等待执行的任务
private final ReentrantLock mainLock = new ReentrantLock();
//线程池的主要状态锁，对线程池状态（比如线程池大小、runState等）的改变都要使用这个锁
private final HashSet<Worker> workers = new HashSet<Worker>();
//用来存放工作集
private volatile long  keepAliveTime;
//线程存货时间
private volatile boolean allowCoreThreadTimeOut;
//是否允许为核心线程设置存活时间
private volatile int   corePoolSize;
//核心池的大小（即线程池中的线程数目大于这个参数时，提交的任务会被放进任务缓存队列）
private volatile int   maximumPoolSize;
//线程池最大能容忍的线程数
private volatile int   poolSize;
//线程池中当前的线程数
private volatile RejectedExecutionHandler handler;
//任务拒绝策略
private volatile ThreadFactory threadFactory;
//线程工厂，用来创建线程
private int largestPoolSize;
//用来记录线程池中曾经出现过的最大线程数
private long completedTaskCount;
//用来记录已经执行完毕的任务个数
```

corePoolSize就是日常线程池大小，maximumPoolSize在我看来是线程池的一种补救措施，即任务量突然过大时的一种补救措施。largestPoolSize只是一个用来起记录作用的变量，用来记录线程池中曾经有过的最大线程数目，跟线程池的容量没有任何关系。

在ThreadPoolExecutor类中，最核心的任务提交方法是execute()方法，虽然通过submit也可以提交任务，但是实际上submit方法里面最终调用的还是execute()方法。

1）首先，要清楚corePoolSize和maximumPoolSize的含义；

2）其次，要知道Worker是用来起到什么作用的；

3）要知道任务提交给线程池之后的处理策略，这里总结一下主要有4点：

- 如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；
- 如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；
- 如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；
- 如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

3 线程池中的线程初始化

在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

prestartCoreThread()：初始化一个核心线程；

prestartAllCoreThreads()：初始化所有核心线程

```
public boolean prestartCoreThread() {
    return addIfUnderCorePoolSize(null); //注意传进去的参数是null
}

public int prestartAllCoreThreads() {
    int n = 0;
    while (addIfUnderCorePoolSize(null))//注意传进去的参数是null
        ++n;
    return n;
}
```

4 任务缓存队列及排队策略

在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。

workQueue的类型为BlockingQueue，通常可以取下面三种类型：

1）ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

2）LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX_VALUE；

3）synchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

5 任务拒绝策略

当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略：

```
ThreadPoolExecutor.AbortPolicy:
// 丢弃任务并抛出RejectedExecutionException异常。
ThreadPoolExecutor.DiscardPolicy：
// 也是丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy：
// 丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.CallerRunsPolicy：
// 由调用线程处理该任务
```

6 线程池的关闭

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

> shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
> shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

7 线程池容量的动态调整

ThreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()，

setCorePoolSize：设置核心池大小
setMaximumPoolSize：设置线程池最大能创建的线程数目大小

使用示例

不过在java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是使用Executors类中提供的几个静态方法来创建线程池：

```
Executors.newCachedThreadPool();
//创建一个缓冲池，缓冲池容量大小\Integer.MAX_VALUE
Executors.newSingleThreadExecutor();
//创建容量为1的缓冲池
Executors.newFixedThreadPool(int);
//创建固定容量大小的缓冲池
```

- newFixedThreadPool创建的线程池corePoolSize和maximumPoolSize值是相等的，它使用的LinkedBlockingQueue；
- newSingleThreadExecutor将corePoolSize和maximumPoolSize都设置为1，也使用的LinkedBlockingQueue；
- newCachedThreadPool将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX_VALUE，使用的SynchronousQueue，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。

实际中，如果Executors提供的三个静态方法能满足要求，就尽量使用它提供的三个方法，因为自己去手动配置ThreadPoolExecutor的参数有点麻烦，要根据实际任务的类型和数量来进行配置。

另外，如果ThreadPoolExecutor达不到要求，可以自己继承ThreadPoolExecutor类进行重写。

如何合理配置线程池的大小

如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 NCPU+1

如果是IO密集型任务，参考值可以设置为2*NCPU

> 参考资料：http://www.importnew.com/19011.html

SYNC和Lock锁机制

用法区别

synchronized：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。

lock：需要显示指定起始位置和终止位置。一般使用**ReentrantLock**类做为锁，多个线程中必须要使用一个ReentrantLock类做为对象才能保证锁的生效。且在加锁和解锁处需要通过lock()和unlock()显示指出。所以一般会在finally块中写unlock()以防死锁。

性能区别

synchronized是托管给JVM执行的，而lock是java写的控制锁的代码。
在Java1.5中，synchronize是性能低效的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用Java提供的Lock对象，性能更高一些。但是到了Java1.6，发生了变化。synchronize在语义上很清晰，可以进行很多优化，有适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在Java1.6上synchronize的性能并不比Lock差。

synchronized原始采用的是CPU**悲观锁**机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。

而Lock用的是**乐观锁**方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作（Compare and Swap）。我们可以进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是compareAndSetState。这里其实就是调用的CPU提供的特殊指令。

synchronized和reentrantlock二者都是**可重入锁**。基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个synchronized方法时，比如说method1，而在method1中会调用另外一个synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2。

在Java中，synchronized就**不是**可中断锁，而Lock**是**可中断锁。如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

- synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；
- 通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。
- Lock可以提高多个线程进行读操作的效率。
- 在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。
- 当需要以下高级特性时，才应该使用Lock：可定时的、可轮询的与可中断的锁获取操作，公平队列，或者非块结构的锁。

线程通信

1. 同步 这里讲的同步是指多个线程通过synchronized关键字这种方式来实现线程间的通信。
2. while轮询 线程A不断地改变条件，线程ThreadB不停地通过while语句检测这个条件(list.size()==5)是否成立 ，从而实现了线程间的通信。但是这种方式会浪费CPU资源。之所以说它浪费资源，是因为JVM调度器将CPU交给线程B执行时，它没做啥“有用”的工作，只是在不断地测试 某个条件是否成立。就类似于现实生活中，某个人一直看着手机屏幕是否有电话来了，而不是： 在干别的事情，当有电话来时，响铃通知TA电话来了。
3. wait/notify机制 我们还可以看到，两个线程都是在同步块中调用的wait()和notify()方法。如果一个线程在没有获得对象锁的前提下调用了这个对象的wait()或notify()方法，方法调用时将会抛出 IllegalMonitorStateException异常。

既然调用对象wait()方法的线程需要获得这个对象的锁，那么这会不会阻塞其它线程调用这个对象的notify()方法呢？答案是不会阻塞，当一个线程调用监控对象的wait()方法时，它便会释放掉这个监控对象锁，以便让其它线程能够调用这个对象的notify()方法或者wait()方法。

1. 管道通信就是使用java.io.PipedInputStream 和 java.io.PipedOutputStream进行通信

分布式系统中说的两种通信机制：共享内存机制和消息通信机制。
感觉前面的①中的synchronized关键字和②中的while轮询 “属于” **共享内存机制**，由于是轮询的条件使用了volatile关键字修饰时，这就表示它们通过判断这个“共享的条件变量“是否改变了，来实现进程间的交流。

而管道通信，更像**消息传递机制**，也就是说：通过管道，将一个线程中的消息发送给另一个。

volatile

Java 内存模型中的可见性、原子性和有序性。

- **可见性**，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果，另一个线程马上就能看到。**用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。**所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。在 Java 中 volatile、synchronized 和 final 实现可见性。

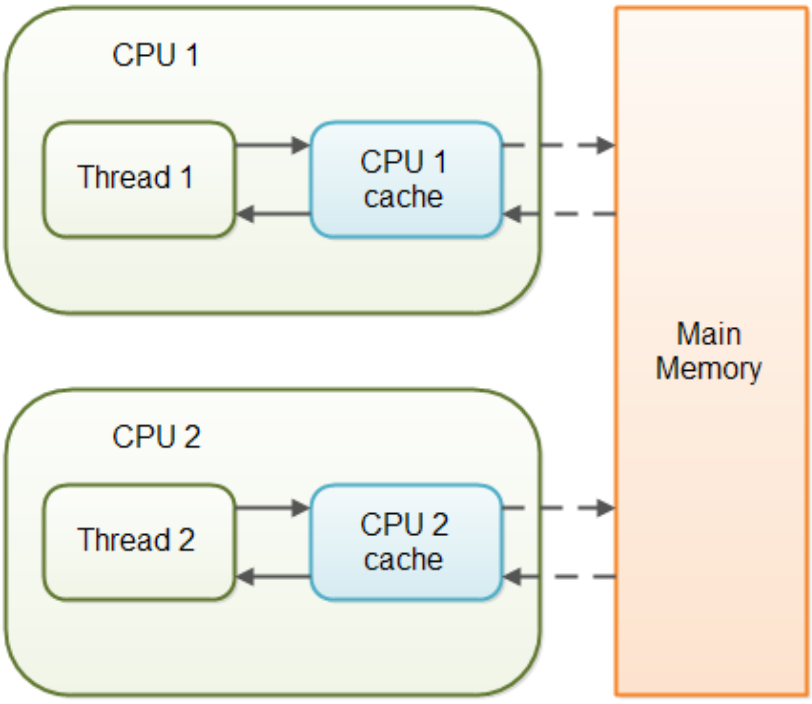
- **原子性：**原子是世界上的最小单位，具有不可分割性。比如 a=0；（a非long和double类型） 这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++； 这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。非原子操作都会存在线程安全问题，需要我们使用同步技术（sychronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

  > 在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

- **有序性：**Java 语言提供了 volatile 和 synchronized 两个关键字来保证线程之间操作的有序性，volatile 是因为其本身包含“禁止指令重排序”的语义，synchronized 是由“一个变量在同一个时刻只允许一条线程对其进行 lock 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

Java语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。

在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比sychronized关键字更轻量级的同步机制。



当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到CPU缓存中。如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。

而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。

当一个变量定义为 volatile 之后，将具备两种特性：

1.保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存（详见：Java内存模型）来完成。

2.禁止指令重排序优化。有volatile修饰的变量，赋值后多执行了一个“load addl $0x0, (%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障；（什么是指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理）。

volatile 性能：

volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

ThreadLocal

先解释一下，在并发编程的时候，成员变量如果不做任何处理其实是线程不安全的，各个线程都在操作同一个变量，显然是不行的，并且我们也知道volatile这个关键字也是不能保证线程安全的。那么在有一种情况之下，我们需要满足这样一个条件：变量是同一个，但是每个线程都使用同一个初始值，也就是使用同一个变量的一个新的副本。这种情况之下ThreadLocal就非常使用，比如说DAO的数据库连接，我们知道DAO是单例的，那么他的属性Connection就不是一个线程安全的变量。而我们每个线程都需要使用他，并且各自使用各自的。这种情况，ThreadLocal就比较好的解决了这个问题。

```
public class ConnectionUtil {
    private static ThreadLocal<Connection> tl = new ThreadLocal<Connection>();
    private static Connection initConn = null;
    static {
        try {
            initConn = DriverManager.getConnection("url, name and password");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public Connection getConn() {
        Connection c = tl.get();
        if(null == c) tl.set(initConn);
        return tl.get();
    }

}
```

这样子，都是用同一个连接，但是每个连接都是新的，是同一个连接的副本。

应用场景：当很多线程需要多次使用同一个对象，并且需要该对象具有相同初始化值的时候最适合使用ThreadLocal。

ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

首先，在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就

是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量）。

初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。

然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找。

最常见的ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。

CyclicBarrier

CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier默认的构造方法是CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。

```
public class CyclicBarrierTest {

    static CyclicBarrier c = new CyclicBarrier(2);

    public static void main(String[] args) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    c.await();
                } catch (Exception e) {

                }
                System.out.println(1);
            }
        }).start();

        try {
            c.await();
        } catch (Exception e) {

        }
        System.out.println(2);
    }
}
```

如果把new CyclicBarrier(2)修改成new CyclicBarrier(3)则主线程和子线程会永远等待，因为没有第三个线程执行await方法，即没有第三个线程到达屏障，所以之前到达屏障的两个线程都不会继续执行。

CyclicBarrier还提供一个更高级的构造函数CyclicBarrier(int parties, Runnable barrierAction)，用于在线程到达屏障时，优先执行barrierAction，方便处理更复杂的业务场景。

```
public class CyclicBarrierTest2 {

    static CyclicBarrier c = new CyclicBarrier(2, new A());

    public static void main(String[] args) {
        new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    c.await();
                } catch (Exception e) {

                }
                System.out.println(1);
            }
        }).start();

        try {
            c.await();
        } catch (Exception e) {

        }
        System.out.println(2);
    }

    static class A implements Runnable {

        @Override
        public void run() {
            System.out.println(3);
        }

    }

}
```

CyclicBarrier的应用场景

CyclicBarrier可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个Excel保存了用户所有银行流水，每个Sheet保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后，再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水。

CyclicBarrier和CountDownLatch的区别

CountDownLatch的计数器只能使用一次。而CyclicBarrier的计数器可以使用reset() 方法重置。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。
CyclicBarrier还提供其他有用的方法，比如getNumberWaiting方法可以获得CyclicBarrier阻塞的线程数量。isBroken方法用来知道阻塞的线程是否被中断。比如以下代码执行完之后会返回true。

CountDownLatch

CyclicBarrier和CountDownLatch一样，都是关于线程的计数器。

直译过来就是倒计数(CountDown)门闩(Latch)。倒计数不用说，门闩的意思顾名思义就是阻止前进。在这里就是指 CountDownLatch.await() 方法在倒计数为0之前会阻塞当前线程。
作用

> java Thread中， join() 方法主要是让调用该方法的thread完成run方法里面的东西后， 再执行join()方法后面的代码。

CountDownLatch 的作用和 Thread.join() 方法类似，可用于一组线程和另外一组线程的协作。例如，主线程在做一项工作之前需要一系列的准备工作，只有这些准备工作都完成，主线程才能继续它的工作。这些准备工作彼此独立，所以可以并发执行以提高速度。在这个场景下就可以使用 CountDownLatch 协调线程之间的调度了。在直接创建线程的年代（Java 5.0 之前），我们可以使用 Thread.join()。在 JUC 出现后，因为线程池中的线程不能直接被引用，所以就必须使用 CountDownLatch 了。

```
class Driver { // ...
    void main() throws InterruptedException {
        CountDownLatch startSignal = new CountDownLatch(1);
        CountDownLatch doneSignal = new CountDownLatch(N);

        for (int i = 0; i < N; ++i) // create and start threads
            new Thread(new Worker(startSignal, doneSignal)).start();

        doSomethingElse();            // don't let run yet
        startSignal.countDown();      // let all threads proceed
        doSomethingElse();
        doneSignal.await();           // wait for all to finish
    }
}

class Worker implements Runnable {
    private final CountDownLatch startSignal;
    private final CountDownLatch doneSignal;
    Worker(CountDownLatch startSignal, CountDownLatch doneSignal) {
        this.startSignal = startSignal;
        this.doneSignal = doneSignal;
    }
    public void run() {
        try {
            startSignal.await();
            doWork();
            doneSignal.countDown();
        } catch (InterruptedException ex) {} // return;
    }

    void doWork() { ... }
}
```

当 startSignal.await() 会阻塞线程，当 startSignal.countDown() 被调用之后，所有 Worker 线程开始执行 doWork() 方法，所以 Worker。doWork() 是几乎同时开始执行的。当 Worker.doWork() 执行完毕后，调用 doneSignal.countDown()，在所有 Worker 线程执行完毕之后，主线程继续执行。

区别：

CountDownLatch 适用于一组线程和另一个主线程之间的工作协作。一个主线程等待一组工作线程的任务完毕才继续它的执行是使用 CountDownLatch 的主要场景；CyclicBarrier 用于一组或几组线程，比如一组线程需要在一个时间点上达成一致，例如同时开始一个工作。另外，CyclicBarrier 的循环特性和构造函数所接受的 Runnable 参数也是 CountDownLatch 所不具备的。

Atom包

Java.util.concurrent中提供了atomic原子包，可以实现原子操作（atomic operation），即在多线程环境中，执行的操作不会被其他线程打断。

Atomic包是Java.util.concurrent下的另一个专门为线程安全设计的Java包，包含多个原子操作类。这个包里面提供了一组原子变量类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由JVM从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。

实际上是借助硬件的相关指令来实现的，不会阻塞线程(**或者说只是在硬件级别上阻塞了**)。可以对**基本数据**、**数组中的基本数据**、**对类中的基本数据**进行操作。原子变量类相当于一种泛化的volatile变量，能够支持原子的和有条件的读-改-写操作。

传统锁的问题

我们先来看一个例子：计数器（Counter），采用Java里比较方便的锁机制synchronized关键字，初步的代码如下：

```
class Counter {

    private int value;

    public synchronized int getValue() {
        return value;
    }

    public synchronized int increment() {
        return ++value;
    }

    public synchronized int decrement() {
        return --value;
    }
}
```

我们需要更有效，更加灵活的机制，synchronized关键字是基于阻塞的锁机制，也就是说当一个线程拥有锁的时候，访问同一资源的其它线程需要等待，直到该线程释放锁。

这里会有些问题：首先，如果被阻塞的线程优先级很高很重要怎么办？其次，如果获得锁的线程一直不释放锁怎么办？（这种情况是非常糟糕的）。还有一种情况，如果有大量的线程来竞争资源，那CPU将会花费大量的时间和资源来处理这些竞争（事实上CPU的主要工作并非这些），同时，还有可能出现一些例如死锁之类的情况，最后，其实锁机制是一种比较粗糙，粒度比较大的机制，相对于像计数器这样的需求有点儿过于笨重，因此，对于这种需求我们期待一种更合适、更高效的线程安全机制。

硬件同步策略

现在的处理器都支持多重处理，当然也包含多个处理器共享外围设备和内存，同时，加强了指令集以支持一些多处理的特殊

需求。特别是几乎所有的处理器都可以将其他处理器阻塞以便更新共享变量。

当前的处理器基本都支持CAS，只不过每个厂家所实现的算法并不一样罢了，每一个CAS操作过程都包含三个运算符：**一个内存地址V，一个期望的值A和一个新值B**，操作的时候如果这个地址上存放的值等于这个期望的值A，则将地址上的值赋为新值B，否则不做任何操作。CAS的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。我们来看一个例子，解释CAS的实现过程（并非真实的CAS实现）：

```
class SimulatedCAS {
    private int value;

    public synchronized int getValue() {
        return value;
    }
    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        int oldValue = value;
        if (value == expectedValue)
            value = newValue;
        return oldValue;
    }
}
```

下面是一个用CAS实现的Counter：

```
public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.getValue();
    }

    public int increment() {
        int oldValue = value.getValue();
        while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)
            oldValue = value.getValue();
        return oldValue + 1;
    }
}
```

Atomic类

标量类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference
数组类：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray
更新器类：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater
复合变量类：AtomicMarkableReference，AtomicStampedReference

- int addAndGet(int delta)：以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果
- boolean compareAndSet(int expect, int update)：如果输入的数值等于预期值，则以原子方式将该值设置为输入的值。
- int getAndIncrement()：以原子方式将当前值加1，注意：这里返回的是自增前的值。
- int getAndSet(int newValue)：以原子方式设置为newValue的值，并返回旧值。

第一组AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference这四种基本类型用来处理布尔，整数，长整数，对象四种数据，其内部实现不是简单的使用synchronized，而是一个更为高效的方式**CAS (compare and swap) + volatile和native方法**，从而避免了synchronized的高开销，执行效率大为提升。

AtomicInteger的实现：

```
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```

这里直接调用一个叫Unsafe的类去处理。

Atomic中的CAS

```
public final int getAndAddInt(Object o, long offset, int delta) {
        int v;
        do {
            v = getIntVolatile(o, offset);//------------0---------------
        } while (!compareAndSwapInt(o, offset, v, v + delta));//-------------1-------------
        return v;
    }

public final native boolean compareAndSwapInt(Object o, long offset,//---------------2--------
------
                                              int expected,
                                              int x);
```

我稍微解释一下，其实compareAndSwapInt的注释解释的很明确，原子的将变量的值更新为x，如果成功了返回true，我们知道，如果我们创建AtomicInteger实例时不传入参数，则原始变量的值即为0，所以上面//———-0———–处得到的v的值即为0,1处的代码为：
while(!compareAndSwapInt(o, offset, 0, 1))我们知道offset指向的地址对应的值就是原始变量的初值0，所以与期望的值0相同，所以将初值赋值为1，返回true，取反后为false，循环结束，返回v即更新之前的值0. 这就是类似于i++操作的原子操作的实现，当然最终CAS的实现都是native的，用C语言实现的，我们这里看不到源码，有时间我会反编译一下这段代码看看。

说了半天，我们要回归到最原始的问题了：这样怎么实现线程安全呢？请大家自己先考虑一下这个问题，其实我们在语言层面是没有做任何同步的操作的，大家也可以看到源码没有任何锁加在上面，可它为什么是线程安全的呢？这就是Atomic包下这些类的奥秘：语言层面不做处理，我们将其交给硬件—CPU和内存，利用CPU的多处理能力，实现硬件层面的阻塞，再加上volatile变量的特性即可实现基于原子操作的线程安全。所以说，CAS并不是无阻塞，只是阻塞并非在语言、线程方面，而是在**硬件层面**，所以无疑这样的操作会更快更高效！

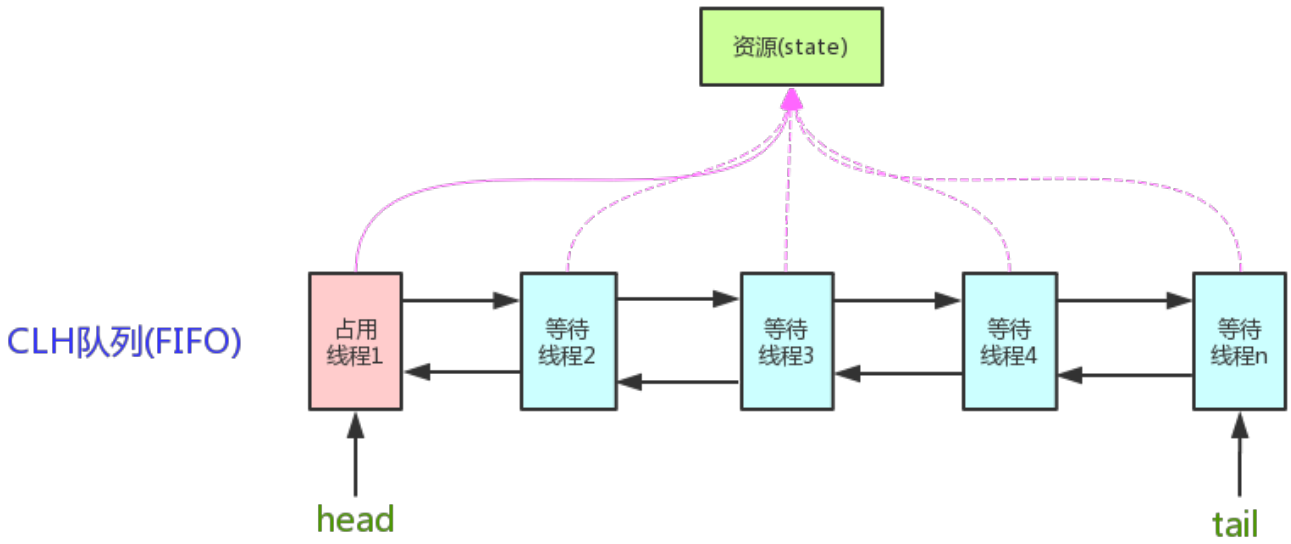
> CAS采用**总线加锁**或**缓存加锁**方式保证原子性。

虽然基于CAS的线程安全机制很好很高效，但要说的是，并非所有线程安全都可以用这样的方法来实现，这只适合一些粒度比较小，型如计数器这样的需求用起来才有效，否则也不会有锁的存在了。

AQS

谈到并发，不得不谈ReentrantLock；而谈到ReentrantLock，不得不谈AbstractQueuedSynchronized（AQS）！

类如其名，抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch…



它维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）。这里volatile是核心关键词，具体volatile的语义，在此不述。state的访问方式有三种:

- getState()
- setState()
- compareAndSetState()

AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

- isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
- tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
- tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
- tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
- tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回true，否则返回false。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

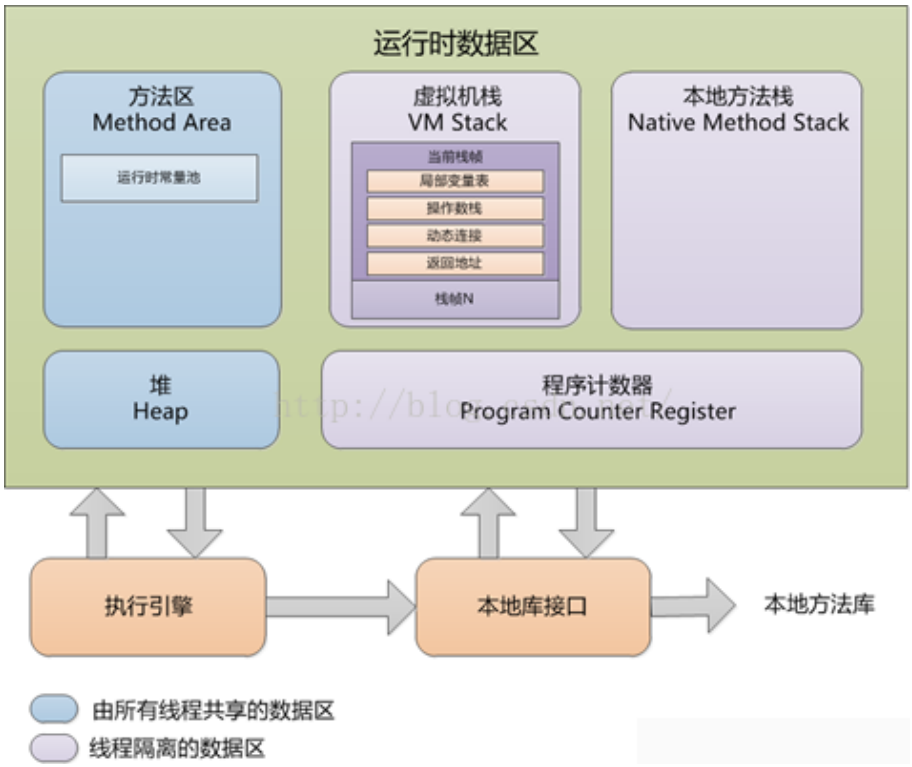
JVM

内存模型、GC垃圾回收、包括分代、GC算法、收集器、类加载和双亲委派、
JVM调优、内存泄漏和内存溢出。

内存模型

JVM定义了若干个程序执行期间使用的数据区域。这个区域里的一些数据在**JVM启动**的时候创建，在**JVM退出**的时候销毁。而其他的数据依赖于每一个线程，在**线程创建**时创建，在**线程退出**时销毁。

> 一个java程序的创建，就是一个进程，对应一个jvm虚拟机，一个java进程有一整套jvm内存模型



线程隔离的数据区

- **程序计数器**

  程序计数器是一块较小的内存空间，可以看作是当前线程所执行的**字节码**的**行号指示器**。

  > 分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

  由于Java 虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“**线程私有**”的内存。

- 如果线程正在执行的是一个Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；
- 如果正在执行的是Natvie 方法，这个计数器值则为空（Undefined）。

> 此内存区域是唯一一个在Java 虚拟机规范中没有规定任何**OutOfMemoryError**情况的区域。

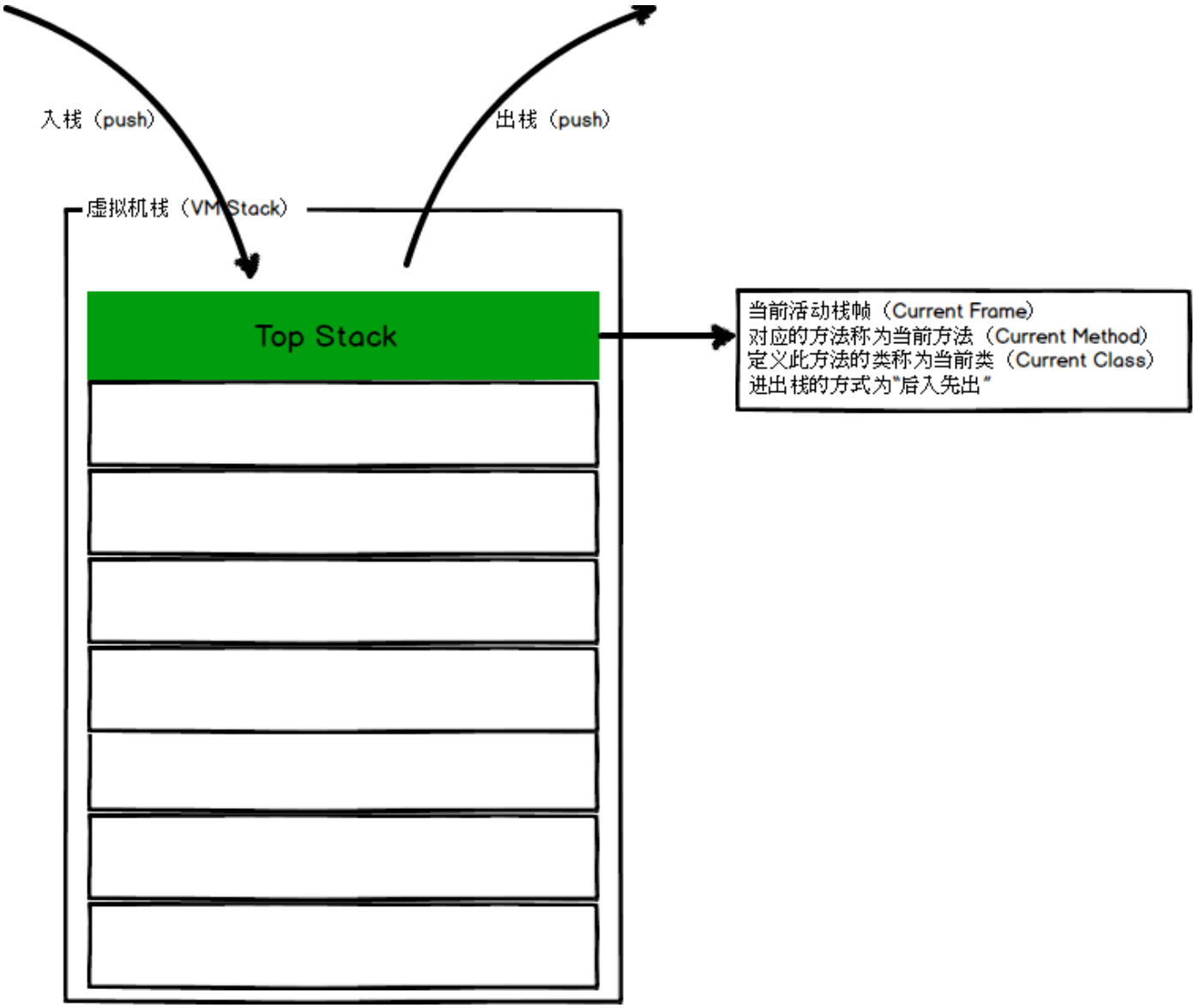
- **虚拟机栈**

每创建一个线程，虚拟机就会为这个线程创建一个虚拟机栈。

线程私有，它的生命周期与线程相同。虚拟机栈描述的是Java 方法执行的内存模型：每个方法被执行的时候都会同时创建**一个栈帧（Stack Frame）**用于存储 局部变量表 、 操作数栈 、 动态链接 、 方法出口 等信息。

动画是由一帧一帧图片连续切换结果的结果而产生的，其实虚拟机的运行和动画也类似，每个在虚拟机中运行的程序也是由许多的帧的切换产生的结果，只是这些帧里面存放的是方法的局部变量，操作数栈，动态链接，方法返回地址和一些额外的附加信息组成。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

> 对于执行引擎来说，活动线程中，只有栈顶的栈帧是有效的，称为**当前栈帧**，这个栈帧所关联的方法称为**当前方法**。执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。



- 局部变量表

  局部变量表是一组变量值存储空间，用于存放**方法参数**和**方法内部定义的局部变量**。在Java程序被编译成Class文件时，就在方法的**Code属性的max_locals数据项**中确定了该方法所需要分配的最大局部变量表的容量。

  局部变量表的容量以变量槽（Slot）为最小单位，32位虚拟机中一个Slot可以存放一个32位以内的数据类型（boolean、byte、char、short、int、float、reference和returnAddress八种）。

  1. **reference**类型虚拟机规范没有明确说明它的长度，但一般来说，虚拟机实现至少都应当能从此引用中直接或者间接地查找到对象在Java堆中的起始地址索引和方法区中的对象类型数据。
  2. **returnAddress**类型是为**字节码指令jsr、jsr_w和ret服务**的，它指向了一条字节码指令的地址。

  虚拟机是使用局部变量表完成参数值到参数变量列表的传递过程的，如果是实例方法（非static），那么局部变量表的第0位索引的Slot默认是用于传递方法所属对象实例的引用，在方法中通过this访问。
  Slot是可以重用的，当Slot中的变量超出了作用域，那么下一次分配Slot的时候，将会覆盖原来的数据。Slot对对象的引用会影响GC（要是被引用，将不会被回收）。
  系统不会为局部变量赋予初始值（实例变量和类变量都会被赋予初始值）。也就是说不存在类变量那样的准备阶段。

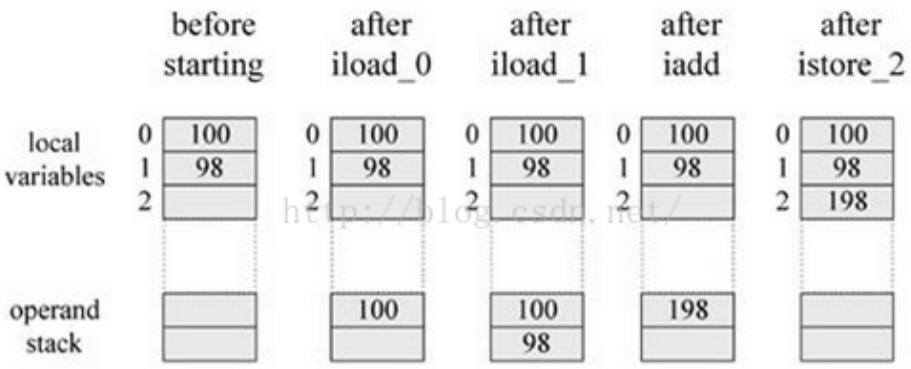
- 操作数栈

  和局部变量区一样，操作数栈也是被组织成一个以字长为单位的数组。但是和前者不同的是，**它不是通过索引来访问，而是通过标准的栈操作——压栈和出栈—来访问的。**比如，如果某个指令把一个值压入到操作数栈中，稍后另一个指令就可以弹出这个值来使用。

  虚拟机在操作数栈中存储数据的方式和在局部变量区中是一样的：如int、long、float、double、reference和returnType的存储。对于byte、short以及char类型的值在压入到操作数栈之前，也会被转换为int。

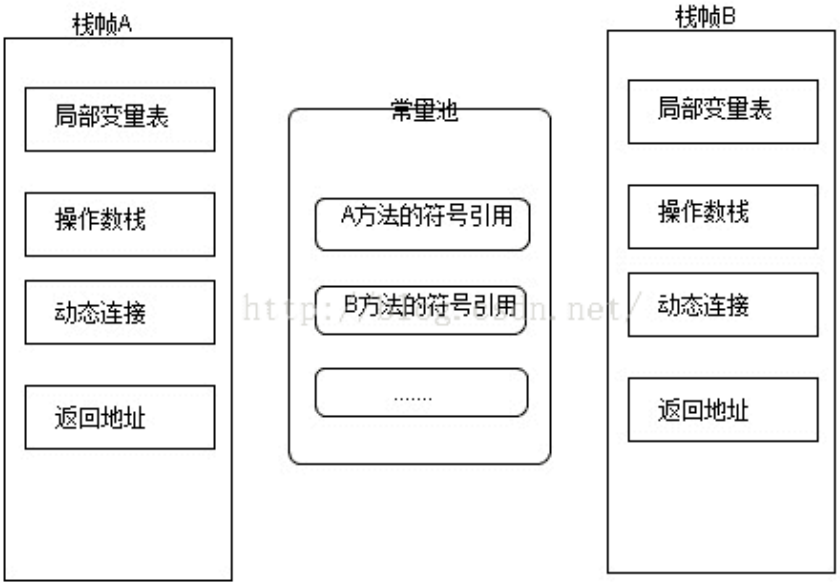
  【实际运算】虚拟机把操作数栈作为它的工作区——大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈。

  在这个字节码序列里，前两个指令iload*0和iload*1将存储在局部变量中索引为0和1的整数压入操作数栈中，其后iadd指令从操作数栈中弹出那两个整数相加，再将结果压入操作数栈。第四条指令istore_2则从操作数栈中弹出结果，并把它存储到局部变量区索引为2的位置。下图详细表述了这个过程中局部变量和操作数栈的状态变化，图中没有使用的局部变量区和操作数栈区域以空白表示。



- 动态链接

虚拟机运行的时候,运行时常量池会保存大量的符号引用，这些符号引用可以看成是每个方法的间接引用。如果代表栈帧A的方法想调用代表栈帧B的方法，那么这个虚拟机的方法调用指令就会以B方法的符号引用作为参数，但是因为符号引用并不是直接指向代表B方法的内存位置，所以在调用之前还必须要将符号引用转换为直接引用，然后通过直接引用才可以访问到真正的方法。
如果符号引用是在类加载阶段或者第一次使用的时候转化为直接应用，那么这种转换成为静态解析，如果是在运行期间转换为直接引用，那么这种转换就成为动态连接。



- 返回地址

  方法的返回分为两种情况，一种是**正常退出**，退出后会根据方法的定义来决定是否要传返回值给上层的调用者，一种是**异常导致的方法结束**，这种情况是不会传返回值给上层的调用方法。

  不过无论是那种方式的方法结束，在退出当前方法时都会跳转到当前方法被调用的位置，如果方法是正常退出的，则调用者的PC计数器的值就可以作为返回地址,，果是因为异常退出的，则是需要通过异常处理表来确定。

  方法的的一次调用就对应着栈帧在虚拟机栈中的一次入栈出栈操作，因此方法退出时可能做的事情包括：恢复上层方法的局部变量表以及操作数栈，如果有返回值的话，就把返回值压入到调用者栈帧的操作数栈中，还会把PC计数器的值调整为方法调用入口的下一条指令。

- 异常

  在Java 虚拟机规范中，对虚拟机栈规定了两种异常状况：如果线程请求的**栈深度大于虚拟机所允许的深度**，将抛出StackOverflowError异常；如果虚拟机栈可以动态扩展（当前大部分的Java 虚拟机都可动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈），**当扩展时无法申请到足够的内存**时会抛出OutOfMemoryError异常。

- **本地方法栈**

本地方法栈（Native MethodStacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java 方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native 方法服务。

> native是与C++联合开发的时候用的！使用native关键字说明这个方法是原生函数，也就是这个方法是用C/C++语言实现的，并且被编译成了DLL，**由java去调用**。 这些函数的实现体在DLL中，JDK的源代码中并不包含，你应该是看不到的。对于不同的平台它们也是不同的。**这也是java的底层机制**，实际上java就是在不同的平台上调用不同的native方法实现对操作系统的访问的。

虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并**没有强制规定**，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

由所有线程共享的数据区

- **堆**（HEAP）

堆是Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的**对象实例**都在这里分配内存。jvm只有一个堆区(heap)被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身。

堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC堆”。

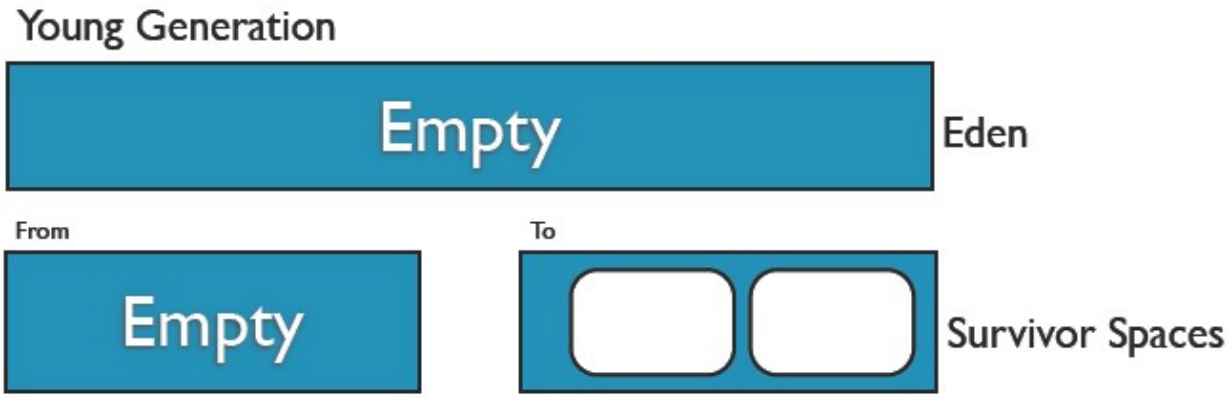
堆的大小可以通过**-Xms(最小值)**和**-Xmx(最大值)**参数设置，-Xms为JVM启动时申请的最小内存，默认为操作系统物理内存的1/64但小于1G，-Xmx为JVM可申请的最大内存，默认为物理内存的1/4但小于1G。

默认当空余堆内存小于40%时，JVM会增大Heap到-Xmx指定的大小，可通过-XX:**MinHeap**FreeRation=来指定这个比列；当空余堆内存大于70%时，JVM会减小heap的大小到-Xms指定的大小，可通过-XX:**MaxHeap**FreeRation=来指定这个比列。

如果从**内存回收**的角度看，由于现在收集器基本都是采用的**分代收集算法**，所以Java 堆中还可以细分为：**新生代和老年代**；

1. **新生代**：程序新创建的对象都是从新生代分配内存，新生代由Eden Space【伊甸园】和两块相同大小的Survivor Space【幸存区】(通常又称S0和S1或From和To)构成，可通过-Xmn参数来指定新生代的大小，也可以通过-XX:SurvivorRation来调整Eden Space及SurvivorSpace的大小。
    - Eden Space字面意思是伊甸园，对象被创建的时候首先放到这个区域，进行垃圾回收后，不能被回收的对象被放入到空的survivor区域。
    - Survivor Space幸存者区，用于保存在eden space内存区域中经过垃圾回收后没有被回收的对象。Survivor有两个，分别为To Survivor、 From Survivor，这个两个区域的空间大小是一样的。执行垃圾回收的时候Eden区域不能被回收的对象被放入到空的survivor（也就是To Survivor，同时Eden区域的内存会在垃圾回收的过程中全部释放），另一个survivor（即From Survivor）里不能被回收的对象也会被放入这个survivor（即To Survivor），然后To Survivor 和 From Survivor的标记会互换，始终保证一个survivor是空的。

    Eden Space和Survivor Space都属于新生代，新生代中执行的垃圾回收被称之为**Minor GC**（因为是对新生代进行垃圾回收，所以又被称为Young GC），每一次**Young GC后留下来的对象age加1**。



2. **老年代**：用于存放经过多次新生代GC仍然存活的对象（age >），例如缓存对象，新建的对象也有可能直接进入老年代，主要有两种情况：1、大对象，可通过启动参数设置-XX:PretenureSizeThreshold=1024(单位为字节，默认为0)来代表超过多大时就不在新生代分配，而是直接在老年代分配。2、大的数组对象，且数组中无引用外部对象。

   > 老年代所占的内存大小为-Xmx对应的值减去-Xmn对应的值。

当老年代被放满的之后，虚拟机会进行垃圾回收，称之为**Major GC**。由于Major GC除并发GC外均需对整个堆进行扫描和回收，因此又称为**Full GC**。

```
如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError 异常。
```

- **方法区**

方法区在一个jvm实例的内部，**类型信息**被存储在一个称为方法区的内存逻辑区中。类型信息是由类加载器在*类加载*时从类文件中提取出来的。**类(静态)变量**也存储在方法区中。

简单说方法区用来存储类型的元数据信息，一个**.class文件**是类被java虚拟机使用之前的表现形式，一旦这个类要被使用，java虚拟机就会对其进行**装载、连接（验证、准备、解析）和初始化**。

而装载后的结果就是由.class文件转变为方法区中的一段特定的**数据结构**。这个数据结构会存储如下信息：

```
类型信息
        这个类型的全限定名
        这个类型的直接超类的全限定名
        这个类型是类类型还是接口类型
        这个类型的访问修饰符
        任何直接超接口的全限定名的有序列表

字段信息
        字段名
        字段类型
        字段的修饰符

方法信息
        方法名
        方法返回类型
        方法参数的数量和类型（按照顺序）
        方法的修饰符

其他信息
        除了常量以外的所有类（静态）变量
        一个指向ClassLoader的指针
        一个指向Class对象的指针
        常量池（常量数据以及对其他类型的符号引用）
```

JVM为每个已加载的类型都维护一个常量池。常量池就是这个类型用到的常量的一个有序集合，包括实际的常量(string,integer,和floating point常量)和对类型，域和**方法的符号引用**。池中的数据项象数组项一样，是通过索引访问的。

每个类的这些元数据，无论是在构建这个类的实例还是调用这个类某个对象的方法，都会访问方法区的这些元数据。

例如：构建一个对象时，JVM会在堆中给对象分配空间，这些空间用来存储当前对象实例属性以及其父类的实例属性（而这些属性信息都是从方法区获得），注意，这里并不是仅仅为当前对象的实例属性分配空间，还需要给父类的实例属性分配，

到此其实我们就可以回答第一个问题了，即实例化父类的某个子类时，JVM也会同时构建父类的一个对象。从另外一个角度也可以印证这个问题：调用当前类的构造方法时，首先会调用其父类的构造方法直到Object，而构造方法的调用意味着实例的创建，所以子类实例化时，父类肯定也会被实例化。

类变量被类的所有实例共享，即使没有类实例时你也可以访问它。这些变量只与类相关，所以在方法区中，它们成为类数据在逻辑上的一部分。在JVM使用一个类之前，它必须在方法区中为每个non-final类变量分配空间。

方法区主要有以下几个特点：
1、方法区是线程安全的。由于所有的线程都共享方法区，所以，方法区里的数据访问必须被设计成线程安全的。例如，假如同时有两个线程都企图访问方法区中的同一个类，而这个类还没有被装入JVM，那么只允许一个线程去装载它，而其它线程必须等待
2、方法区的大小不必是固定的，JVM可根据应用需要动态调整。同时，方法区也不一定是连续的，方法区可以在一个堆(甚至是JVM自己的堆)中自由分配。
3、方法区也可被垃圾收集，当某个类不在被使用(不可触及)时，JVM将卸载这个类，进行垃圾收集

> 可以通过-XX:PermSize 和 -XX:MaxPermSize 参数限制方法区的大小。

相对而言，垃圾收集行为在这个区域是比较少出现的，但并非数据进入了方法区就如永久代的名字一样“永久”存在了。这个区域的内存回收目标主要是针对**常量池的回收**和对**类型的卸载**。

当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

总结

总结

名称	特征	作用	配置参数	异常
程序计数器	占用内存小，线程私有， 生命周期与线程相同	大致为字节码行号指示器	无	无
虚拟机栈	线程私有，生命周期与线程相同，使用连续的内存空间	Java 方法执行的内存模型，存储局部变量表、操作栈、动态链接、方法出口等信息	-Xss	StackOverflowError OutOfMemoryError
java堆	线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址	保存对象实例，所有对象实例（包括数组）都要在堆上分配	-Xms -Xsx -Xmn	OutOfMemoryError
方法区	线程共享，生命周期与虚拟机相同，可以不使用连续的内存地址	存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据	-XX:PermSize:16M -XX:MaxPermSize64M	OutOfMemoryError
运行时常量池	方法区的一部分，具有动态性	存放字面量及符号引用		

直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致OutOfMemoryError 异常出现，所以我们放到这里一起讲解。
在JDK 1.4 中新加入了NIO（NewInput/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O 方式，它可以使用Native 函数库直接分配堆外内存，然后通过一个存储在Java 堆里面的DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java 堆和Native 堆中来回复制数据。

堆与栈的对比

经常有人把Java 内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java内存区域的划分实际上远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。
堆很灵活，但是不安全。对于对象，我们要动态地创建、销毁，不能说后创建的对象没有销毁，先前创建的对象就不能销毁，那样的话我们的程序就寸步难行，所以Java中用堆来存储对象。而一旦堆中的对象被销毁，我们继续引用这个对象的话，就会出现著名的 NullPointerException，这就是堆的缺点——错误的引用逻辑只有在运行时才会被发现。
栈不灵活，但是很严格，是安全的，易于管理。因为只要上面的引用没有销毁，下面引用就一定还在，在大部分程序中，都是先定义的变量、引用先进栈，后定义的后进栈，同时，区块内部的变量、引用在进入区块时压栈，区块结束时出栈，理解了这种机制，我们就可以很方便地理解各种编程语言的作用域的概念了，同时这也是栈的优点——错误的引用逻辑在编译时就可以被发现。
栈–主要存放引用和基本数据类型。
堆–用来存放 new 出来的对象实例。

内存溢出和内存泄漏

内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer，但给它存了long才能存下的数，那就是内存溢出。
内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。
memory leak会最终会导致out ofmemory。

内存分配过程

1、JVM 会试图为相关Java对象在Eden Space中初始化一块内存区域。
2、当Eden空间足够时，内存申请结束；否则到下一步。
3、JVM 试图释放在Eden中所有不活跃的对象（这属于1或更高级的垃圾回收）。释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区。
4、Survivor区被用来作为Eden及Old的中间交换区域，当Old区空间足够时，Survivor区的对象会被移到Old区，否则会被保留在Survivor区。
5、当Old区空间不够时，JVM 会在Old区进行完全的垃圾收集（0级）。
6、完全垃圾收集后，若Survivor及Old区仍然无法存放从Eden复制过来的部分对象，导致JVM无法在Eden区为新对象创建内存区域，则出现“outofmemory”错误。

> 参考链接：http://blog.csdn.net/u012152619/article/details/46968883

GC垃圾回收

- **垃圾回收机制的意义**

  Java语言中一个显著的特点就是引入了java回收机制，是c++程序员最头疼的内存管理的问题迎刃而解，它使得java程序员在编写程序的时候不在考虑**内存管理**。由于有个垃圾回收机制，java中的额对象不在有“作用域”的概念，只有**对象的引用**才有“作用域”。垃圾回收可以有效的**防止内存泄露**，有效的使用空闲的内存。

  > **内存泄露**：指该内存空间使用完毕后未回收，在不涉及复杂数据结构的一般情况下，java的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有是也将其称为“对象游离”.

- **垃圾回收机制中的算法**

Java语言规范没有明确地说明JVM使用哪种垃圾回收算法，但是任何一种**垃圾回收算法**一般要做2件基本的事情：
（a）**发现无用信息对象**；（判断对象是垃圾）
（b）回收被无用对象占用的内存空间，使该空间可被程序再次使用。

一、发现无用信息对象

1. **引用计数法(Reference Counting Collector)**

   引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个**引用计数**。
   a) 当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为1。
   b) 当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1）；
   c) 但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。
   d) 任何引用计数器为0的对象实例可以被当作垃圾收集。
   e) 当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

   优点：

   引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

   缺点：

   无法检测出**循环引用**。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0.

   ```
   public class Main {
       public static void main(String[] args) {
           MyObject object1 = new MyObject();
           MyObject object2 = new MyObject();

           object1.object = object2;
           object2.object = object1;

           object1 = null;
           object2 = null;
       }
   }
   ```

   最后面两句将object1和object2赋值为null，也就是说object1和object2指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数器都不为0，那么垃圾收集器就永远不会回收它们。

   > 调用System.gc()可人工进行垃圾回收。

2. **tracing算法(Tracing Collector) 【可达性分析算法】**

   现在主流的商用程序语言（JAVA,C#,Lisp）的主流实现中，都是通过可达性分析来**判断对象**是否存活的。
   根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。

   java中可作为GC Root的对象有

   - 虚拟机栈中引用的对象（本地变量表）

- 方法区中静态属性引用的对象
- 方法区中常量引用的对象
- 本地方法栈中引用的对象（Native对象）

> 无论通过引用计数算法判断对象的引用数量，还是可达性分析算法判断对象的引用链是否科大，判定对象是否存活与**“引用”**有关。

一个对象在传统的定义下只有**被引用**或者**没有被引用**两种状态。
JDK 1.2之后，Java对引用的概念进行了扩充，将引用分为**强引用（Strong）**、**软引用（Soft）**、**弱引用（Weak）**、**虚引用(Phantom)**。

- **强引用**：普遍存在，例如"Object obj = new Object()“这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。
- **软引用**：用来描述非必需的对象。在系统将要发生内存溢出异常之前，将这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，则抛出内存溢出异常。JDK1.2之后，使用SoftReference类来实现软引用。
- **弱引用**：也是用来描述非必需对象。强度小于软引用，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。无论内存是否足够，当垃圾收集器工作时，都会回收掉只被弱引用关联的对象。提供了WeakReference来实现弱引用。JDK1.2之后，使用WeakReference类来实现软引用。
- **虚引用**：也称之为"幽灵引用"或者"幻影引用"，是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。唯一目的就是能在这个垃圾收集器回收的时候，收到一个**系统通知**。JDK1.2之后，使用PhantomReference类来实现软引用。

> 另外，即使是可达性分析算法中不可达的对象，也并非是非死不可的。要真正宣告死亡，至少要经理两次标记过程。第一次：筛选对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都“视为没有必要执行”。如果被判定为有必要执行，就把对象放入F-Queue队列中，执行低优先级的Finalizer执行，而相应触发的finalize()方法是对象逃脱死亡的最后一次机会，称之为第二次标记，在finalize()中成功拯救自己，只要重新与引用链上的任何一个对象简历关联即可，如This。
>
> 现在用的少！

3 . **方法区回收**

方法区（HotSpot虚拟机中的**永久代**）也是有垃圾收集的。但是“性价比”没有在堆中进行垃圾回收来的高。在堆中，尤其是在新生代，常规应用进行一次垃圾收集一般可以回收70%~95%的空间，而永久代的垃圾收集效率远低于此。

永久代的垃圾收集主要回收两部分内容：**废弃常量**和**无用的类**。

- 回收废弃常量与回收Java堆中的对象非常类似。常量池中的其他类（接口）、方法、字段的符号引用类似，没有其他地方引用这个常量的话，就应该被清理出去。
- 满足一下3个条件才能算是“**无用的类**”
  - 该类所有的实例都已经被回收，也就是Java堆中不存在该类的任何实例。
  - 加载该类的ClassLoader已经被回收。
  - 该类对应的java.lang.class对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法。

二、垃圾收集算法

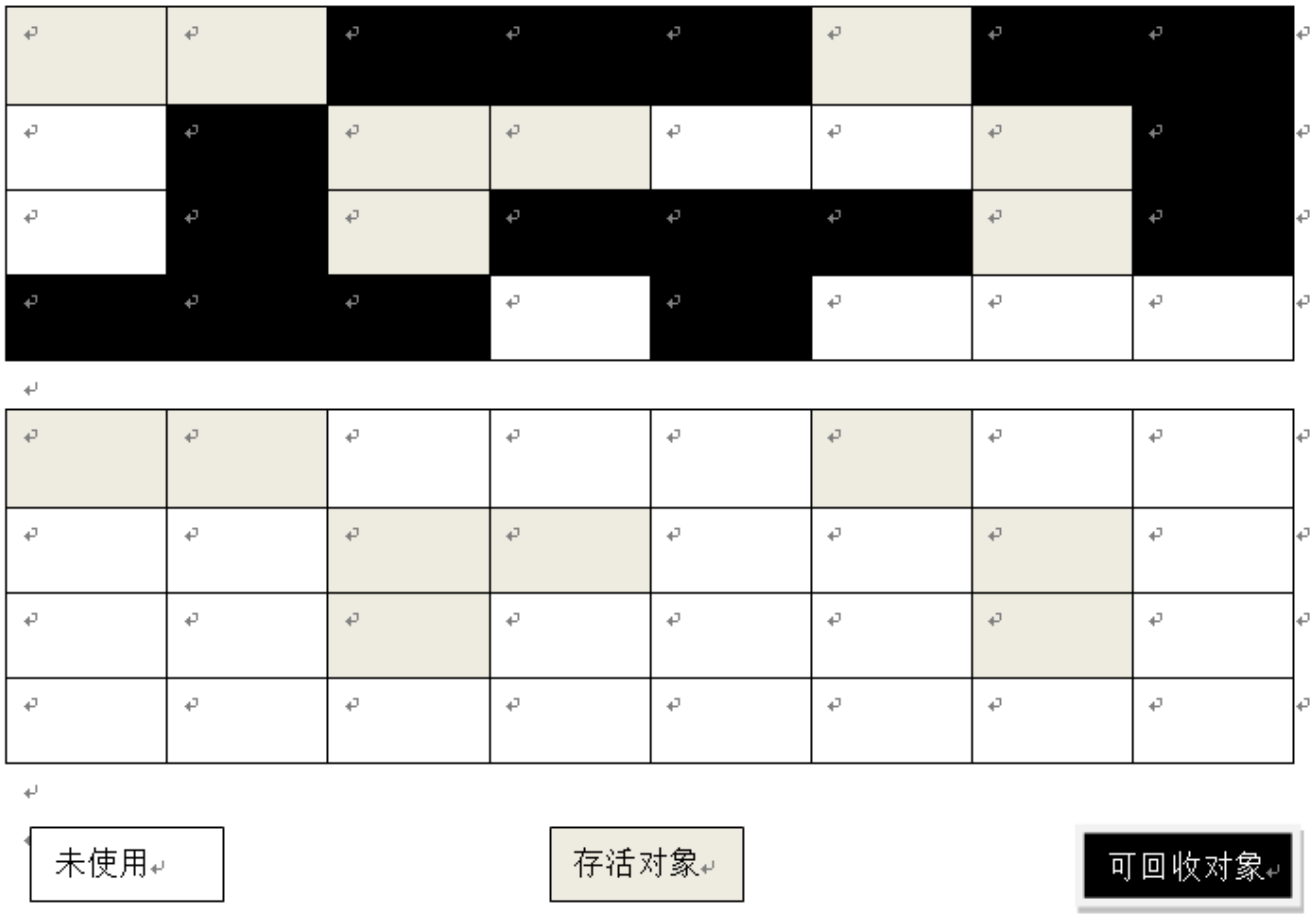
1. **标记-清除算法 (Mark-Sweep)** 最基础

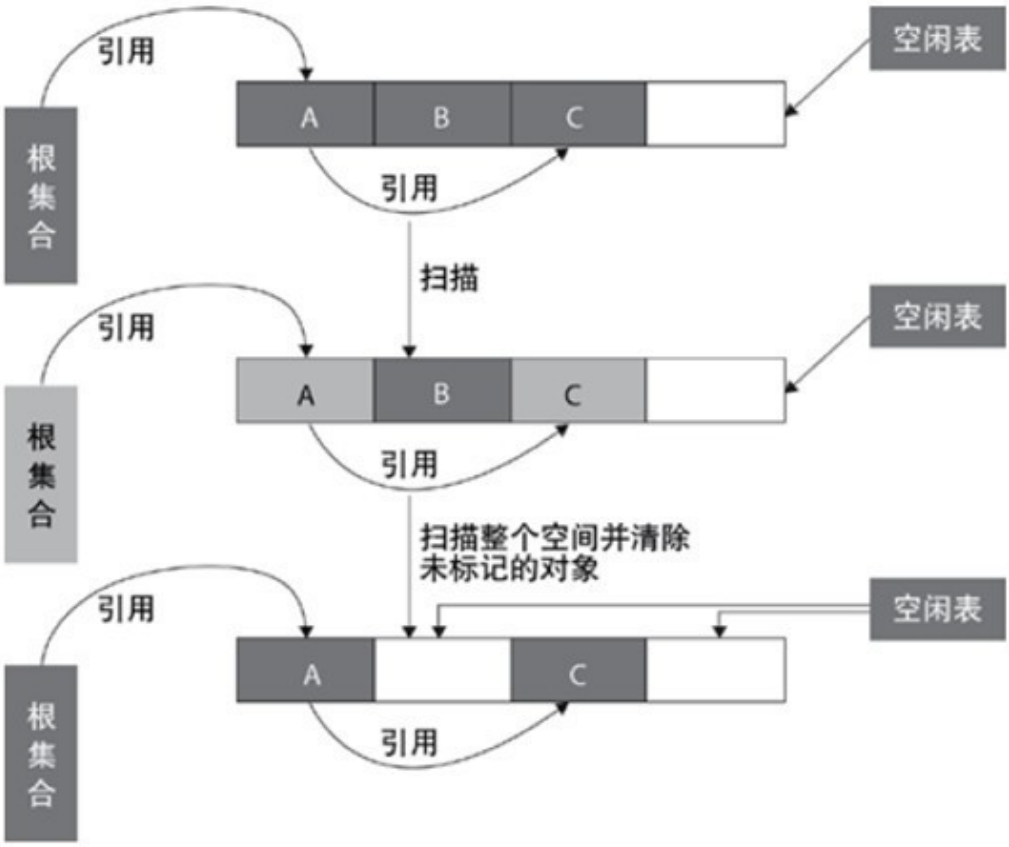
算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成之后回收所有被标记的对象。它的标记过程其实在前一节讲述对象标记判定时已经介绍过了。

标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，在存活对象比较多的情况下极为高效。

不足之处：

- 效率问题：标记和清除两个过程的效率都不高。
- 空间问题：标记清除之后会产生大量不连续的内存碎片，空间碎片太多会导致以后需要分配较大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。





2. **复制算法（Copying）**

   为了解决效率问题，一种被称为”复制“的收集算法出现了，它将可用内存按容量划分成大小相等的两块，每次只使用其中的一块。当这块的内存用完了，就将还存活的对象复制到另外一块上面，然后再把已经使用过得内存一次清理掉。

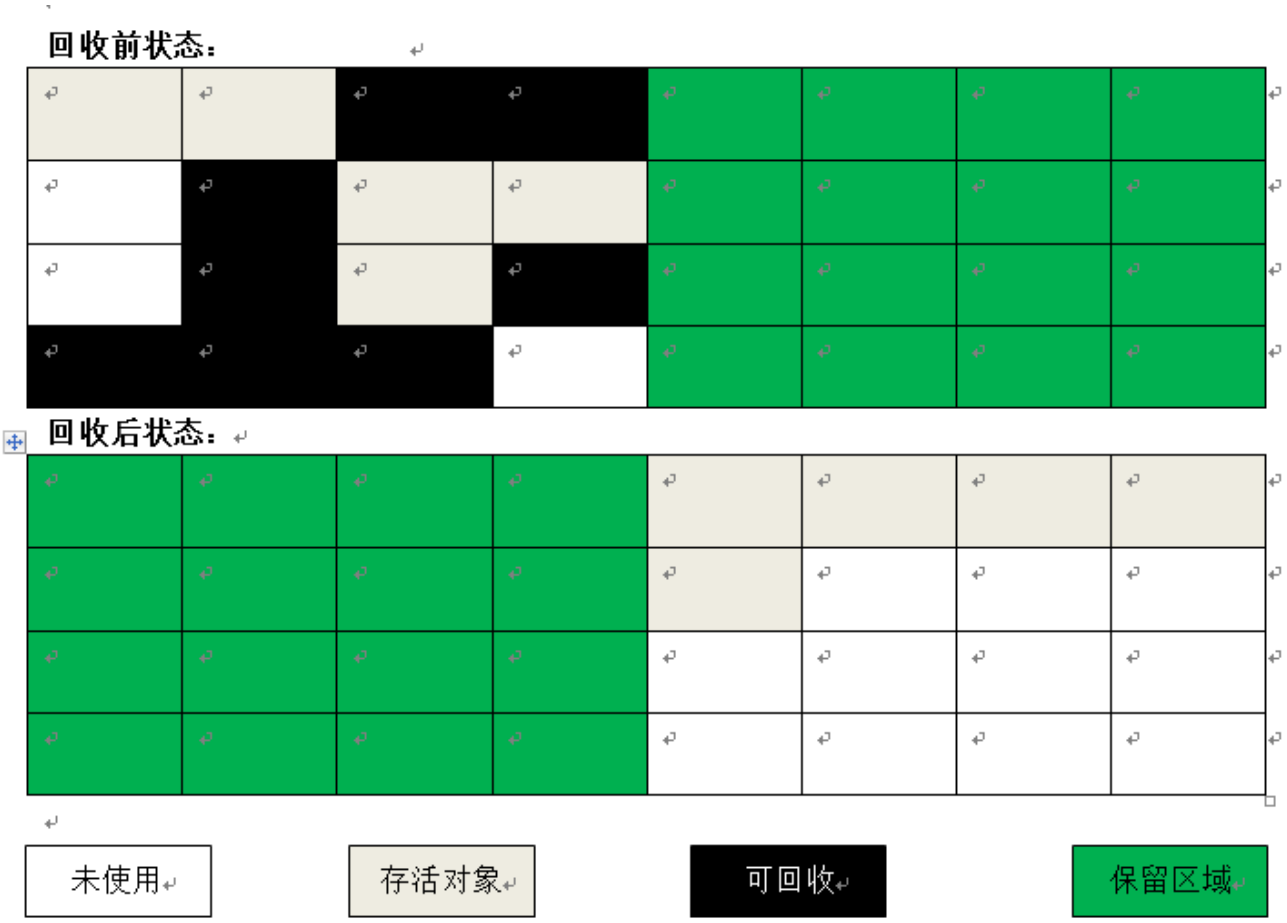
   这样使得每次都对搬去进行内存回收，内存分配的时候不用考虑内容碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

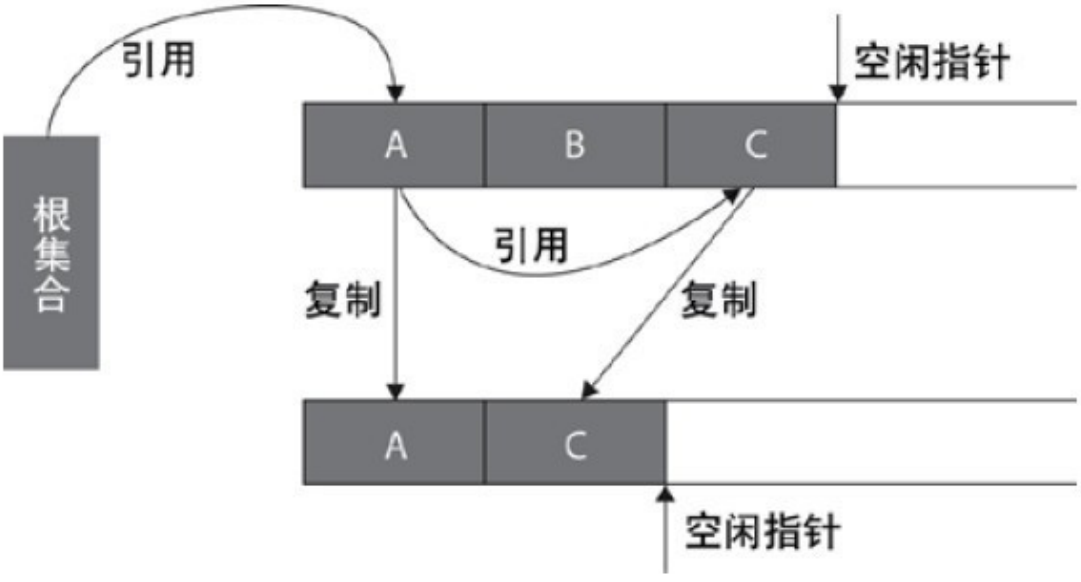
   代价： 将内存缩小为了原来的一半，未免太高了点。

   现在的商业虚拟机都采用这种收集算法来回收**新生代**。研究表明，新生代对象的98%都是“朝生夕死”的，所以不需要按照1：1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两个较小的Survivor空间，每次使用Eden和其中一块Survivor空间。

   当回收时，将Eden和Survivor中还存活的对象一次性地复制到另外一块Survivor空间上最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8：1，也就是每次新生代中可用的空间为整个新生代容量的90%，只有10%的内存是被”浪费“的。

   当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存货，当Survivor空间不够用时，需要依赖其他内存（指老年代）进行**分配担保**。



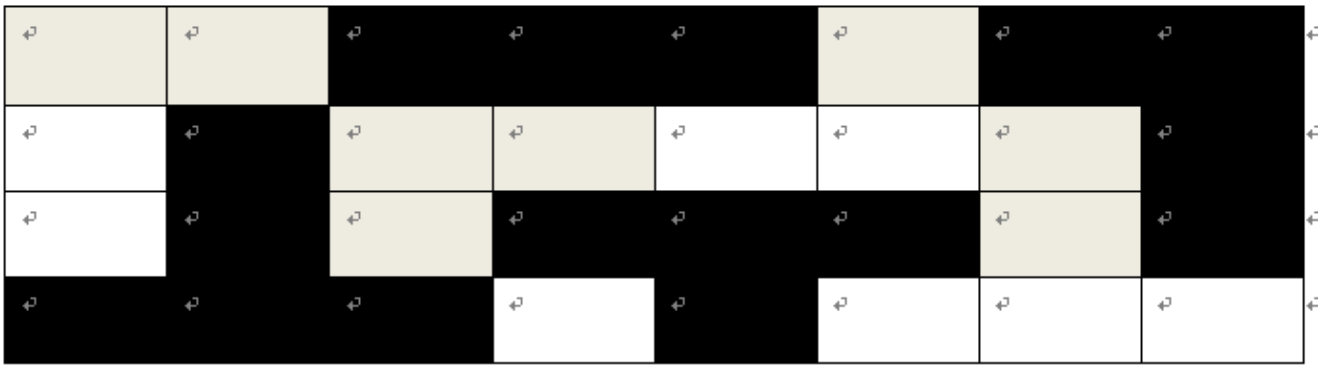


3. **标记-整理算法 （Mark-Compact）**

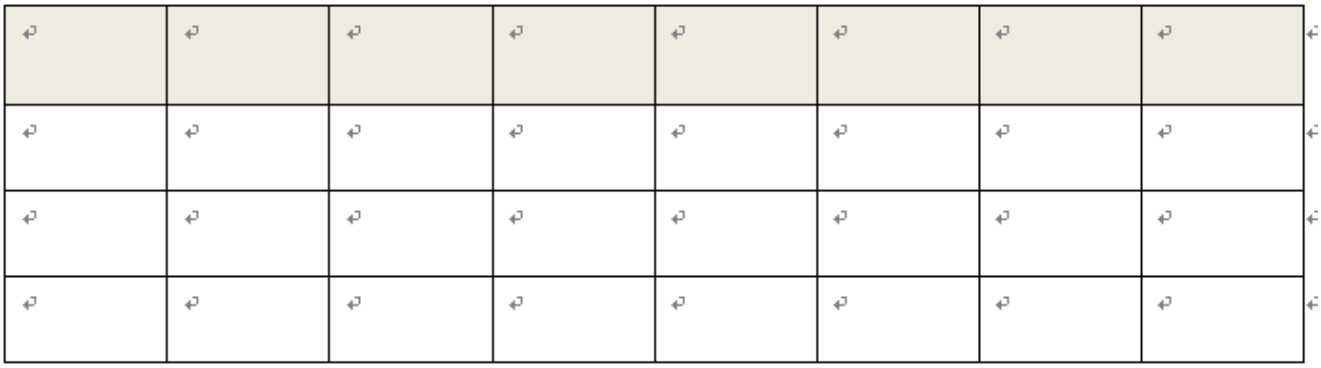
   复制收集算法在对象存活率较高的时候就要进行较多的复制操作，**效率**将会变得很低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行**分配担保**，以应对被使用的内存中所有对象都100%存活的极端情况，所有在老年代一般不能直接使用这种算法。

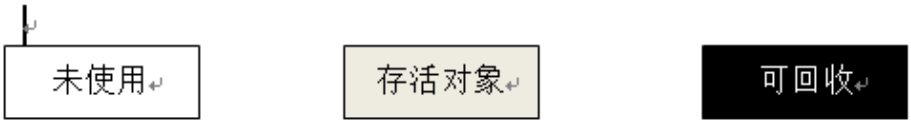
   根据老年代的特点，提出“标记-整理”算法，标记过程仍然与”标记-清除“算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界意外的内存。

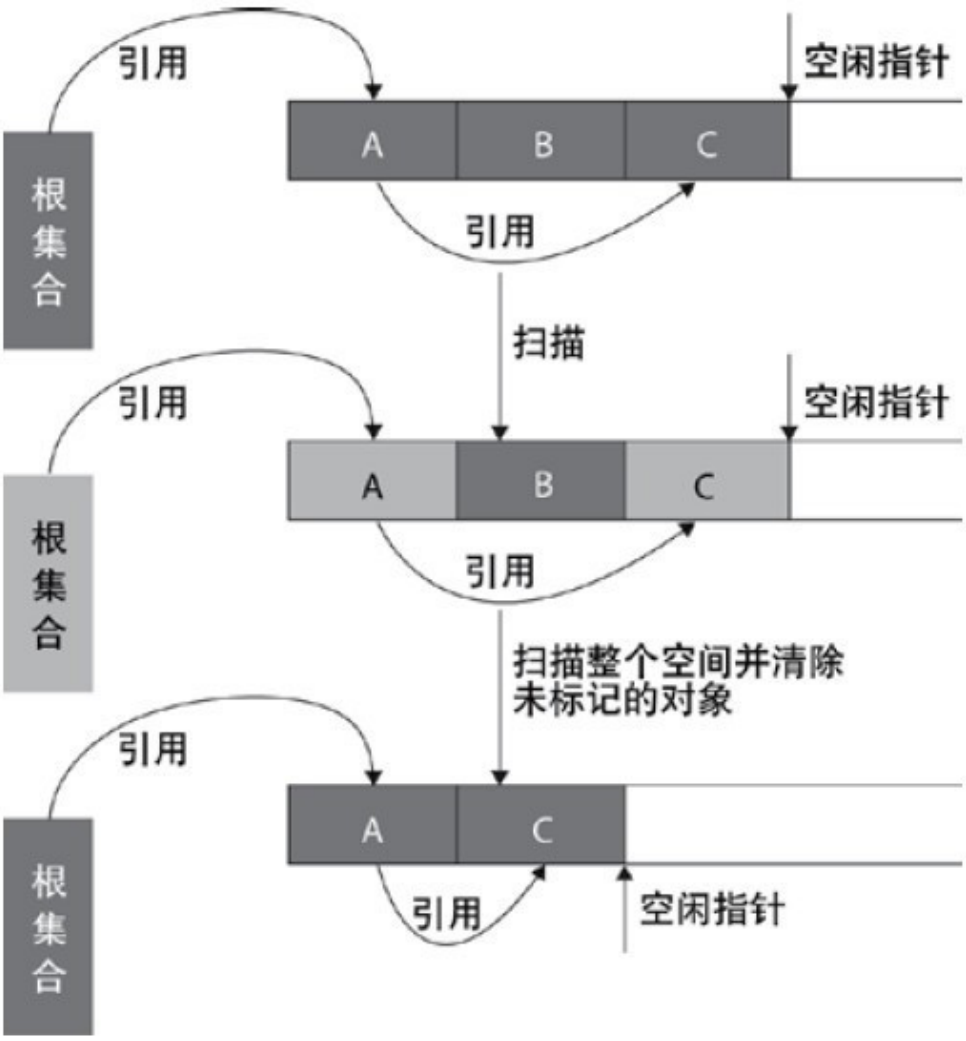
回收前状态：



回收后状态：







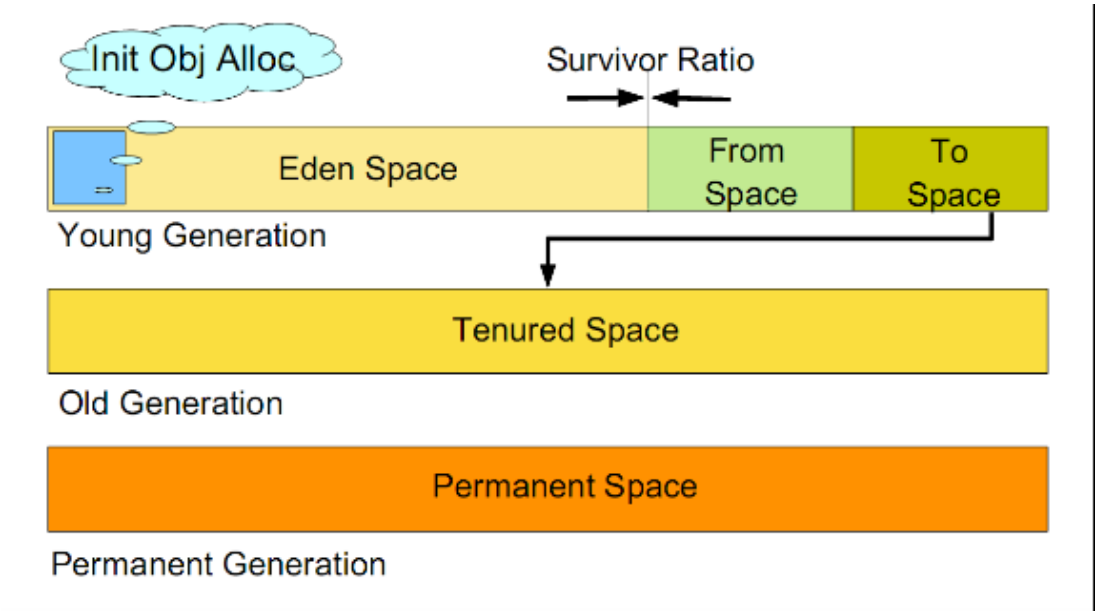
4. **分代收集算法 (Generational Collection)**

   当前商业虚拟机的垃圾收集都采用“分代收集”算法，这种算法没有什么新思想，只是根据对象存活周期的不同将内存划分为几块。一半把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适合的收集算法。

   - **新生代**：每次垃圾回收时都发现有大批对象死去，只有少量存活，那就选用**复制算法**，只需要付出少量存货对象的复制成本就可以完成收集。
   - **老年代**：因对象存活率高，没有额外空间对它进行担保，就必须使用”**标记-清理**“或”**标记-整理**“算法来进行回收。

> 参考资料：http://www.cnblogs.com/andy-zcx/p/5522836.html

分代



由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。
GC有两种类型：**Scavenge GC**和**Full GC**。

- **Scavenge GC**

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。

因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

- **Full GC**

对整个堆进行整理，包括Young、Tenured和Perm。Full GC因为需要对整个对进行回收，所以比Scavenge GC要慢，因此应该尽可能减少Full GC的次数。

在对JVM调优的过程中，很大一部分工作就是对于FullGC的调节。
有如下原因可能导致Full GC：
· 年老代（Tenured）被写满
· 持久代（Perm）被写满
· System.gc()被显示调用

HotSpot的算法实现

- 枚举根节点（HotSpot OopMap）

  OopMap用于枚举GC Roots； OopMap 记录了栈上本地变量到堆上对象的引用关系。其作用是：垃圾收集时，收集线程会对栈上的内存进行扫描，看看哪些位置存储了 Reference 类型。如果发现某个位置确实存的是 Reference 类型，就意味着它所引用的对象这一次**不能**被回收。

  但**问题**是，栈上的本地变量表里面只有一部分数据是 Reference 类型的（它们是我们所需要的），那些非 Reference 类型的数据对我们而言毫无用处，但我们还是不得不对整个栈全部扫描一遍，这是对时间和资源的一种浪费。

  一个很自然的想法是，能不能用空间换时间，在某个时候把栈上代表引用的位置全部记录下来，这样到真正 gc 的时候

就可以直接读取，而不用再一点一点的扫描了。事实上，大部分主流的虚拟机也正是这么做的，比如 HotSpot ，它使用一种叫做 **OopMap** 的数据结构来记录这类信息。

我们知道，一个线程意味着一个栈，一个栈由多个栈帧组成，一个栈帧对应着一个方法，一个方法里面可能有多个安全点。 gc 发生时，程序首先运行到最近的一个**安全点**停下来，然后更新自己的 OopMap ，记下栈上哪些位置代表着引用。枚举根节点时，递归遍历每个栈帧的 OopMap ，通过栈中记录的被引用对象的内存地址，即可找到这些对象（ GC Roots ）。

目前主流的Java虚拟机使用的都是准确式GC，当执行系统停顿下来以后，是有办法直接得知哪些地方存放着对象引用的。在类加载完成的时候，和JIT（Java-In-Time）编译过程中，也会再特定的位置记录下栈和寄存器中哪些位置是引用。

- 安全点 SafePoint

在OopMap的协助下，HotSpot可以快速且准确地完成GC Roots枚举，但可能会导致，引用关系变化，或者说OopMap内容变化的指令非常多，如果为每一条指令都生成对应的OopMap，那将会需要大量的额外空间，这样的GC的空间成本将会变得很高。

HotSpot只在特定位置记录这些信息，这个位置称之为安全点（SafePoint），即程序执行时并非在所有地方都能停顿下来开始GC，只有在到达安全点时才能暂停。

SafePoint的选定既不能太少以至于让GC等待时间太长，也不能过于频繁以致于过分增大运行时的负荷。

这些特定的位置主要在：
1、循环的末尾
2、方法临返回前 / 调用方法的call指令后
3、可能抛异常的位置

对于Sefepoint，另一个需要考虑的问题是如何在GC发生时让所有线程（这里不包括执行JNI调用的线程）都“跑”到最近的安全点上再停顿下来。这里有两种方案可供选择：**抢先式中断**（Preemptive Suspension）和**主动式中断**（Voluntary Suspension）。

抢先式中断：不需要线程的执行代码主动去配合，在GC发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。

主动式中断的思想是当GC需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方。

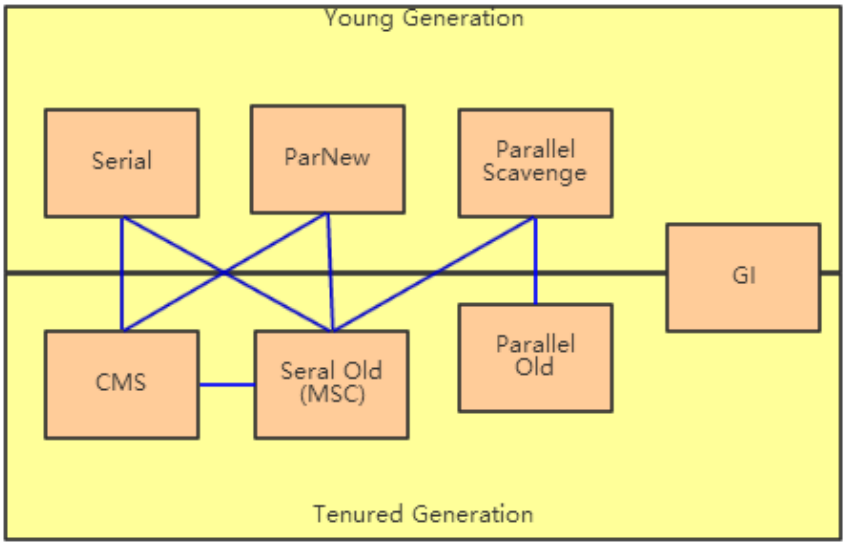
- 安全区域 Safe Region

Safepoint只能处理正在运行的线程，它们可以主动运行到safepoint。而一些Sleep或者被blocked的线程（程序“不执行”的时候）不能主动运行到safepoint。这些线程也需要在GC的时候被标记检查，JVM引入了safe region的概念。
安全区域是指在一段代码片段之中，引用关系不会发生变化。在这个区域中的任意地方开始GC都是安全的。我们也可以把Safe Region看做是被扩展了的SafePoint。

垃圾收集器

JVM中垃圾的回收由垃圾收集器进行，随着JDK的不断升级，垃圾收集器也开发出了各种版本，垃圾收集器不断优化的动力，就是为了实现更短的停顿。

下面是7种不同的分代收集器，如果两个收集器之间有连线，则表示它们之间可以搭配使用；所处的区域表示属于新生代还是老年代收集器。用户根据自己的应用特点和要求组合出各个年代所使用的收集器。



Serial收集器

Serial收集器曾经是虚拟机新生代收集的唯一选择，是一个单线程的收集器，在进行收集垃圾时，必须stop the world，**它是虚拟机运行在Client模式下的默认新生代收集器**。

简单而高效，对于限定单个CPU的环境来说，Serial收集器由于没有现成交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代，停顿时间完全可以控制在几十毫秒最多100毫秒以内，只要不是频繁发生，还是可以接受的。

> Serial采用的是**“复制”**算法对应Serial Old是老年代的收集器，使用**“标记-整理”**算法暂停所有用户线程。

ParNew收集器

ParNew收集器是Serial收集器的**多线程**版本，许多运行在Server模式下的虚拟机中首选的新生代收集器，除Serial外，只有它能与CMS收集器配合工作。

继承了Serial的所有控制参数、收集算法、Stop the World、对象分配原则、回收策略等。

ParNew收集器在单CPU环境中绝对不会有比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百地保证可以超越Serial收集器。但随着CPU数量的增加，它对于GC时系统资源的有效利用还是有好处的。

Parallel Scavenge 收集器

Parallel Scavenge收集器也是新生代收集器，使用复制算法又是并行的多线程收集器，它的目标是达到一个可控制的运行用户代码跟（运行用户代码+垃圾收集时间）的百分比值，即**吞吐量**。是一个吞吐量优先的收集器。也是使用**复制算法**。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台计算而不需要太多交互的任务。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是

- 最大垃圾收集停顿时间 -XX：MaxGCPauseMillis参数

设置值为一个大于0的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。GC停顿时间越短是以牺牲吞吐量和新生代空间换区的，系统把新生代空间调小一点，但是垃圾收集也更频繁一些，同时吞吐量也下降了。

- 吞吐量大小 -XX：GCTimeRatio

  设置值应该是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，相当于吞吐量的倒数。如果参数设为19，那允许的最大GC时间就占5%，如果设为99，就是允许最大1%的垃圾收集时间。

另外，该收集器还有一个参数 -XX：+UseAdaptiveSizePolicy 值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小 (-Xmn) 、Eden和Survivor区的比例 (-XX:SurvivorRatio) 、晋升老年代对象大小 (-XX:PretenureSizeThreshold) 等细节参数了。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。

所以，只需要设置基本的内存数据（-Xmx 设置最大堆），然后使用MaxGCPauseMills参数（更关注停顿时长）或GCTimeRatio（更关注吞吐量）参数给虚拟机设立一个优化目标，具体细节参数的调节工作就交给虚拟机完成。自适应调节策略也是Parallel Scanvenge收集器与ParNew收集器的一个重要区别。

Seral Old收集器

Serial Old是Serial收集器的老年代版本，同样是单线程收集器，使用标记整理算法。这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

如果在Server模式下，那么它有两大用途：
1. JDK 1.5以及之前的版本中与Parallel Scavenge收集器搭配使用。
2. 作为CMS收集器的备选方案，在并发收集发生Concurrent Mode Failure时使用。

Parallel Old收集器

Parrallel Old是Parallel Scanvenge收集器的老年代版本，使用多线程和“标记-整理”算法。

在Parrallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合，在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scanvenge 加 Parallel Old收集器。

CMS 收集器（Concurrent Mark Sweep）

CMS收集器是一种以获得最短回收停顿时间为目标的收集器，基于”标记-清除“算法。

目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用需求。

整个过程分为4个步骤，包括：

- 初始标记 (CMS initial mark)
- 并发标记 (CMS concurrent mark)
- 重新标记 (CMS remark)
- 并发清除 (CMS concurrent sweep)

其中，初始标记、重新标记两个步骤仍然需要Stop the World。**初始标记**仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，**并发标记**阶段就是进行GC Roots Tracing的过程。而**重新标记**阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。所以体现的优先也很明显，**并发收集、低停顿**。

它也有以下三个明显的缺点：

- **CMS收集器对CPU资源非常敏感。**其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，虽然不会造成用户线程停顿，但是会因为占用了一部分线程而导致应用程序变慢，总吞吐量变低。CMS默认启动的回收线程数是（CPU数量 + 3）/ 4，CPU较少的时候CPU负载很大一部分就要去执行收集线程，可能导致用户程序的执行速度降低。后来提出的“增量式并发收集器”i-CMS利用单CPU年代的抢占式模拟多任务机制的思想，但最终还是被淘汰了。

- **CMS收集器无法处理浮动垃圾，可能会出现“Concurrent Mode Failure”失败而导致一次Full GC的产生。**由于并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，只好留待下一次GC时再清理掉。这一部分垃圾称之为“浮动垃圾”。

  也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等待老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。

  通过调整参数：**-XX：CMSInitiatingOccupancyFraction**的值来出发百分比。68%有充足的预留内存空间，92%的话，会导致预留内存无法满足程序需要，就会出现一次concurrent mode failure失败，这是虚拟机就会启动后背预案：临时启动Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以这个参数设置得太高很容易导致大量的Concurrent Mode Failure失败，兴性能反而降低。

- **CMS是一款基于“标记-清除”算法实现的收集器**，会导致大量空间碎片产生，空间碎片过多时，会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前出发一次Full GC。为了解决这个问题，CMS收集器提供了 -XX：+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS收集器顶不住要进行FullGC时，开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片就没有了，但停顿的时间不得不边变长。

虚拟机设计者还提供了另外一个参数 -XX：CMSFullGCsBeforeCompaction，这个参数是用于设置执行多少次不压缩的Full GC之后，跟着来一次带压缩的（默认为0，表示每次进入Full GC都要进行碎片整理）。

G1 收集器 （Garbege First）

G1收集器是当今收集器技术发展的最前沿成果之一，被视为JDK 1.7中的HotSpot虚拟机的一个重要金华特征。

G1是一款面向服务端应用的垃圾收集器。HotSpot开发团队赋予它的使命是未来可以替换掉JDK 1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点：

- **并行与并发**：G1充分利用了多CPU、多核环境下的硬件优势，来缩短Stop-the-World停顿的时间，G1收集器仍然可以通过并发的方式让Java程序继续执行。
- **分代收集**：分代概念依然保留，不需要其他收集器配合就能独立管理整个GC堆，但他能够采用不同的方式去处理新创建的对象和以存活一段时间、熬过多次GC的对象以获得更好的收集效果。
- **空间整合**：基于“标记-整理”算法实现的收集器，不会产生内存空间碎片。有利于长时间运行，分配大对象时，不会因为无法找到连续内存而提前出发下一次GC。
- **可预测的停顿**：这是G1相对于CMS的另一大优势，都是共同的关注点，但G1除了追求低停顿外，还能简历可预测的停顿时间模型，能让使用者明确指定一个长度有M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。他把Java堆划分成多个大小相等的

独立区域（Region），保留了新生代、老年代的概念，但不再是物理隔离的，它们都是一部分Region（不需要连续）的集合。

G1之所以能够建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需要的时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（Garbege First名称的由来。）**这种Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。**

在G1收集器中，**Region之间的对象引用**以及其他收集器中的**新生代与老年代之间的对象**引用，虚拟机都是使用**Remembered Set**来避免全堆扫描的。G1中每个Region都有一个与之对应的Remembered Set，虚拟机发现程序在对Reference类型的数据进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用的对象是否处于不同的Region之中，（在分代的例子中就是检查是否老年代中的对象引用了新生代的对象）如果是，便通过CardTable把相关的引用信息记录到被引用对象所属的Region的Remembered Set之中。当进行内存回收的时候，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏。

G1收集器的运作大致过程是：

- **初始标记**：初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS （Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象，这阶段需要停顿，但耗时很短。
- **并发标记**：GC root的tracing过程，这阶段耗时较长，但可与用户程序并发执行。
- **最终标记**：为了修正并发标记期间因用户程序继续运作而导致标记发生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程Remembered Set中，这阶段需要停止线程，但可并行执行。
- **筛选回收**：最后在筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户期望的GC停顿时间来制定回收计划。可并行，但因为停顿是用户可控制的，而且只回收一部分Region，所以停顿用户线程将大幅提高收集效率。

垃圾收集器参数总结

- **UseSerialGC**：虚拟机运行在Client模式下的默认值，打开此开关后，使用Serial+Serial Old的收集器组合进行内存回收
- **UseParNewGC**：打开此开关后，使用ParNew+Serial Old的收集器组合进行内存回收
- **UseConcMarkSweepGC**：打开此开关后，使用ParNew+CMS+Serial Old的收集器组合进行内存回收。Serial Old收集器将作为CMS收集器出现Concurrent Mode Failure失败后的后备收集器使用
- **UseParallelGC**：虚拟机运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
- **UseParallelOldGC**：打开此开关后，使用Parallel Scavenge + Parallel Old的收集器组合进行内存回收
- **SurvivorRatio**：新生代中Eden区域与Survivor区域的容量比值，默认值为8，代表Eden：Survivor=8:1
- **PretenureSizeThreshold**:直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
- **MaxTenuringThreshold**:晋升到老年代的对象年龄，每个对象在坚持过一次Minor GC之后，年龄就增加1，当超过这个参数时就进入老年代
- **UseAdaptiveSizePolicy**:动态调整Java堆中各个区域的大小以及进入老年代的年龄
- **HandlePromotionFailure**:是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个Eden和Survivor区的

所有对象都存活的极端情况

- **ParallelGCThreads** :设置并行GC时进行内存回收的线程数
- **GCTimeRatio**:GC时间占总时间的比率，默认值为99，即允许1%的GC时间。仅在使用Parallel Scavenge收集器时生效
- **MaxGCPauseMillis**: 设置GC的最大停顿时间，仅在使用Parallel Scavenge收集器时生效
- **CMSInitingOccupancyFraction** :设置CMS收集器在老年代空间被使用多少后触发垃圾收集。默认值为68%，仅在使用CMS收集器时生效
- **UseCMSCompactAtFullCollection**：设置CMS收集器在完成垃圾收集后是否要进行一次内存碎片整理，仅在使用CMS收集器时生效
- **CMSFullGCsBeforeCompaction**：设置CMS收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用CMS收集器时生效

内存分配与回收策略

1. **对象优先在Eden分配**
2. **大对象直接进入老年代**
3. **长期存活的对象将进入老年代**
4. **动态对象年龄判定：**如果Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等待MaxTenuringThreshold中要求的年龄。
5. **空间分配担保**：在发生Minor GC之前，虚拟机会先检查老年大最大可用的连续空间是否大于新生代所有对象的总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置为不允许冒险，那这时也要改为进行一次Full GC。

JVM调优

调优方法
一切都是为了这一步，调优，在调优之前，我们需要记住下面的原则：

1、多数的Java应用不需要在服务器上进行GC优化；
2、多数导致GC问题的Java应用，都不是因为我们参数设置错误，而是代码问题；
3、在应用上线之前，先考虑将机器的JVM参数设置到最优（最适合）；
4、减少创建对象的数量；
5、减少使用全局变量和大对象；
6、GC优化是到最后不得已才采用的手段；
7、在实际使用中，分析GC情况优化代码比优化GC参数要多得多；

GC优化的目的有两个：
1、将转移到老年代的对象数量降低到最小；
2、减少full GC的执行时间；

为了达到上面的目的，一般地，你需要做的事情有：

1、减少使用全局变量和大对象；
2、调整新生代的大小到最合适；
3、设置老年代的大小为最合适；
4、选择合适的GC收集器；

JDK本身提供了很多方便的JVM性能调优监控工具，除了集成式的 【 VisualVM 】 和 【 jConsole 】 外，还有jps、jstack、jmap、jhat、jstat等小巧的工具。

jps：主要用来输出JVM中运行的进程状态信息。
jstack：主要用来查看某个Java进程内的线程堆栈信息。
jmap（Memory Map）和jhat（java Heap Analysis Tool）：jmap用来查看堆内存使用状况，一般结合jhat使用。
jstat（JVM统计监测工具）
jinfo（java配置信息工具）

上面这些调优工具都提供了强大的功能，但是总的来说一般分为以下几类功能：

- 堆信息查看

查看堆空间大小分配，提供即时的垃圾回收功能，垃圾监控，查看堆内类、对象信息查看：数量、类型等，对象引用情况查看

有了**堆信息查看**方面的功能，我们一般可以顺利解决以下问题：
–年老代年轻代大小划分是否合理
–内存泄漏
–垃圾回收算法设置是否合理

- 线程监控

线程信息监控：系统线程数量。
线程状态监控：各个线程都处在什么样的状态下
Dump线程详细信息：查看线程内部运行情况
死锁检查

> **Dump**：JVM 能够记录下问题发生时系统的运行状态并将其存储在转储(dump)文件中，从而为我们分析和诊断问题提供了重要的依据。

- 热点分析

CPU热点：检查系统哪些方法占用的大量CPU时间
内存热点：检查哪些对象在系统中数量最大（一定时间内存活对象和销毁对象一起统计）

这两个东西对于系统优化很有帮助。我们可以根据找到的热点，有针对性的进行系统的瓶颈查找和进行系统优化，而不是漫无目的的进行所有代码的优化。

内存泄漏和内存溢出

内存泄漏是比较常见的问题，而且解决方法也比较通用，这里可以重点说一下，而线程、热点方面的问题则是具体问题具体分析了。

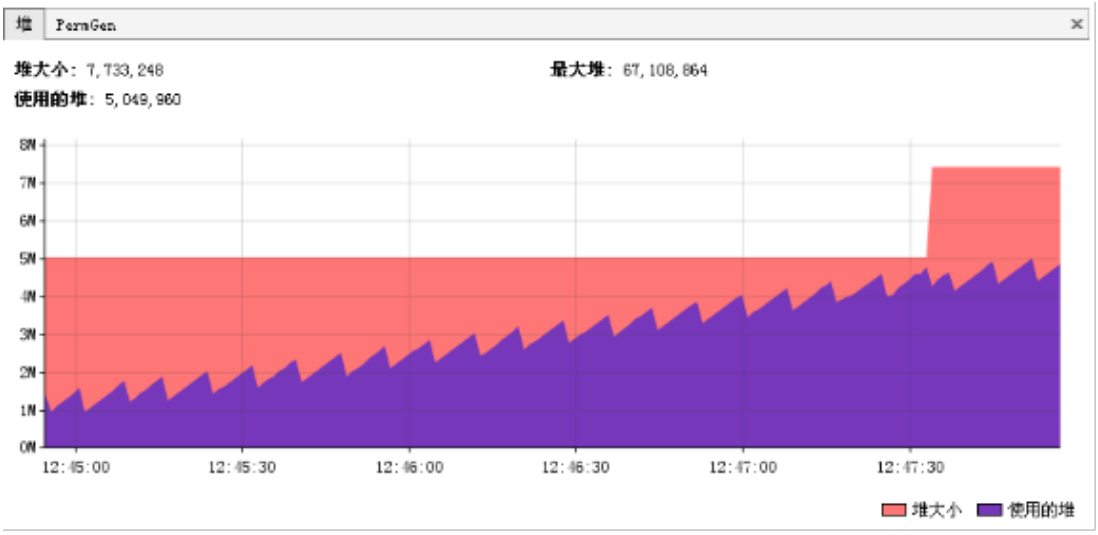
内存泄漏一般可以理解为系统资源（各方面的资源，堆、栈、线程等）在错误使用的情况下，导致使用完毕的资源无法回收（或没有回收），从而导致新的资源分配请求无法完成，引起系统错误。

内存泄漏对系统危害比较大，因为他可以直接导致系统的崩溃。

需要区别一下，内存泄漏和系统超负荷两者是有区别的，虽然可能导致的最终结果是一样的。内存泄漏是用完的资源没有回收引起错误，而系统超负荷则是系统确实没有那么多资源可以分配了（其他的资源都在使用）。

内存泄漏原因以及解决方案

年老代堆空间被占满
异常： java.lang.OutOfMemoryError: Java heap space
说明：



这是最典型的内存泄漏方式，简单说就是所有堆空间都被无法回收的垃圾对象占满，虚拟机无法再在分配新空间。

如上图所示，这是非常典型的内存泄漏的垃圾回收情况图。所有峰值部分都是一次垃圾回收点，所有谷底部分表示是一次垃圾回收后剩余的内存。连接所有谷底的点，可以发现一条由底到高的线，这说明，随时间的推移，系统的堆空间被不断占满，最终会占满整个堆空间。因此可以初步认为系统内部可能有内存泄漏。（上面的图仅供示例，在实际情况下收集数据的时间需要更长，比如几个小时或者几天）。

解决：
这种方式解决起来也比较容易，一般就是根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

持久代被占满
异常：java.lang.OutOfMemoryError: PermGen space
说明：
Perm空间被占满。无法为新的class分配存储空间而引发的异常。这个异常以前是没有的，但是在Java反射大量使用的今天这个异常比较常见了。主要原因就是大量动态反射生成的类不断被加载，最终导致Perm区被占满。
更可怕的是，不同的classLoader即便使用了相同的类，但是都会对其进行加载，相当于同一个东西，如果有N个classLoader那么他将会被加载N次。因此，某些情况下，这个问题基本视为无解。当然，存在大量classLoader和大量反射类的情况其实也不多。

解决：
1. -XX:MaxPermSize=16m
2. 换用JDK。比如JRocket。

堆栈溢出

异常：java.lang.StackOverflowError
说明：这个就不多说了，一般就是递归没返回，或者循环调用造成

线程堆栈满
异常：Fatal: Stack size too small
说明：java中一个线程的空间大小是有限制的。JDK5.0以后这个值是1M。与这个线程相关的数据将会保存在其中。但是当线程空间满了以后，将会出现上面异常。

解决：增加线程栈大小。-Xss2m。但这个配置无法解决根本问题，还要看代码部分是否有造成泄漏的部分。

系统内存被占满
异常：java.lang.OutOfMemoryError: unable to create new native thread
说明：
这个异常是由于操作系统没有足够的资源来产生这个线程造成的。系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大到一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，就出现这个异常了。
分配给Java虚拟机的内存愈多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比的关系。同时，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总共内生产的线程数。
解决：
1. 重新设计系统减少线程数量。
2. 线程数量不能减少的情况下，通过-Xss减小单个线程大小。以便能生产更多的线程。

类加载和双亲委派

与那些在编译时需要进行连接工作的语言不通，在Java语言里面，类型的**加载[1]、连接[2]和初始化[3]**过程都是在程序运行期间完成的，这种策略虽然会令类加载时稍微增加一些**性能开销**，但是会为Java应用程序提供**高度的灵活性**，Java里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。

类加载的时机

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：

1. 加载
2. 验证
3. 准备
4. 解析: 234属于连接
5. 初始化
6. 使用
7. 卸载

七个阶段。它们开始的顺序如下图所示：

什么情况下需要开始类加载过程的第一个阶段：加载?

1. 遇到new、getstatic、putstatic 或 invokestatic 这4条字节码指令时，如果类没有进行过初始化，则需要县触发其初始化。**即**实例化对象的时候、读取或设置一个类的静态字段的时候，以及调用一个类的静态方法的时候。
2. 使用Java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时候，用户需要制定一个要制定的主类，虚拟机会先初始化这个主类。
5. 当使用JDK 1.7的动态语言支持时，如果一个java.lang.invoke.MehodHandle实例最后的解析结果REF*getStatic*、*REF*putStatic、REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

这五种场景中的行为称为对一个类进行主动引用。除此之外，所有引用类的方式都不会触发初始化，称为被动引用。

被动引用：

- 通过子类引用父类的静态字段，不会导致子类初始化。
- 通过数组定义来引用类，不会触发此类的初始化。
- 常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。

注意的是：接口的加载过程和类的加载过程稍有一些不同，针对接口需要做一些特殊说明：接口也有初始化过程，这点和类是一致的，上面的代码都是用静态语句块“static{}”来输出初始化信息的，而接口是不可以使用静态代码块的。

当一个类在初始化时，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不要求其父接口全部都完成了初始化，只有在真正使用到父接口的时候才会初始化。

加载

加载时类加载过程的第一个阶段，在加载阶段，虚拟机需要完成以下三件事情：
1、通过一个类的**全限定名**来获取其定义的二进制字节流。
2、将这个字节流所代表的**静态存储结构**转化为**方法区**的运行时数据结构。
3、在Java堆中生成一个代表这个类的**java.lang.Class**对象，作为对方法区中这些数据的访问入口。

> 这里第1条中的二进制字节流并不只是单纯地从Class文件中获取，比如它还可以从Jar包中获取、从网络中获取（最典型的应用便是Applet）、由其他文件生成（JSP应用）、反射动态代理技术等。

相对于类加载的其他阶段而言，加载阶段（准确地说，是加载阶段**获取类的二进制字节流**的动作）是可控性最强的阶段，因为开发人员既可以使用系统提供的引导类加载器来完成加载，也可以定义自己的类加载器来控制字节流的获取方式（即重写

一个类加载器的loadClass()方法）。

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中（虚拟机未规定此区域的具体数据结构），而且在Java堆中也创建一个java.lang.Class类的对象，这样便可以通过该对象访问方法区中的这些数据。

加载阶段与连接阶段的部分内容是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始，但这些夹在夹在阶段之中进行的动作，仍然属于连接阶段的内容，

验证 【连接一】

验证是连接阶段的第一步，验证的目的是为了确保Class文件中的字节流包含的信息**符合当前虚拟机的要求**，而且不会危害虚拟机自身的安全。

不同的虚拟机对类验证的实现可能会有所不同，但大致都会完成以下四个阶段的验证：**文件格式的验证**、**元数据的验证**、**字节码验证和符号引用验证**。

- **文件格式的验证**：验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机处理，该验证的主要目的是保证输入的字节流能正确地解析并存储于方法区之内。经过该阶段的验证后，字节流才会进入内存的方法区中进行存储，后面的三个验证都是基于方法区的存储结构进行的。
- **元数据验证**：对类的元数据信息进行语义校验（其实就是对类中的各数据类型进行语法校验），保证不存在不符合**Java语法规范**的元数据信息。
- **字节码验证**：该阶段验证的主要工作是进行**数据流**和**控制流**分析，对类的方法体进行校验分析，以保证被校验的类的方法在运行时不会做出危害虚拟机安全的行为。
- **符号引用验证**：这是最后一个阶段的验证，它发生在虚拟机将符号引用转化为直接引用的时候（**解析阶段**中发生该转化，后面会有讲解），主要是对类自身以外的信息（常量池中的各种符号引用）进行匹配性的校验。

准备 【连接二】

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中分配。

1、这时候进行内存分配的仅包括类变量（static），而不包括实例变量，实例变量会在对象实例化时随着对象一块分配在Java堆中。
2、这里所设置的初始值通常情况下是数据类型默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值。

假设一个类变量的定义为：

```
public static int value = 3;
```

那么变量value在准备阶段过后的初始值为0，而不是3，因为这时候尚未开始执行任何Java方法，而把value赋值为3的putstatic指令是在程序编译后，存放于类构造器（）方法之中的，所以把value赋值为3的动作将在**初始化**阶段才会执行。

基本数据类型的零值：

数据类型	默认零值
int	0
long	0L
short	(short)0
char	'\u0000'
byte	(byte)0
boolean	false
float	0.0f
double	0.0d
reference	null

- 对基本数据类型来说，对于类变量（static）和全局变量，如果不显式地对其赋值而直接使用，则系统会为其赋予默认的零值，而对于局部变量来说，在使用前必须显式地为其赋值，否则编译时不通过。
- 对于同时被static和final修饰的常量，必须在声明的时候就为其显式地赋值，否则编译时不通过；而只被final修饰的常量则既可以在声明时显式地为其赋值，也可以在类初始化时显式地为其赋值，总之，在使用前必须为其显式地赋值，系统不会为其赋予默认零值。
- 对于引用数据类型reference来说，如数组引用、对象引用等，如果没有对其进行显式地赋值而直接使用，系统都会为其赋予默认的零值，即null。
- 如果在数组初始化时没有对数组中的各元素赋值，那么其中的元素将根据对应的数据类型而被赋予默认的零值。

如果类字段的字段属性表中存在ConstantValue属性，即同时被final和static修饰，那么在准备阶段变量value就会被初始化为ConstValue属性所指定的值。

假设上面的类变量value被定义为：

```
public static final int value = 3;
```

编译时Javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value赋值为3。

解析 【连接三】

解析阶段是虚拟机将**常量池**中的**符号引用**转化为**直接引用**的过程。

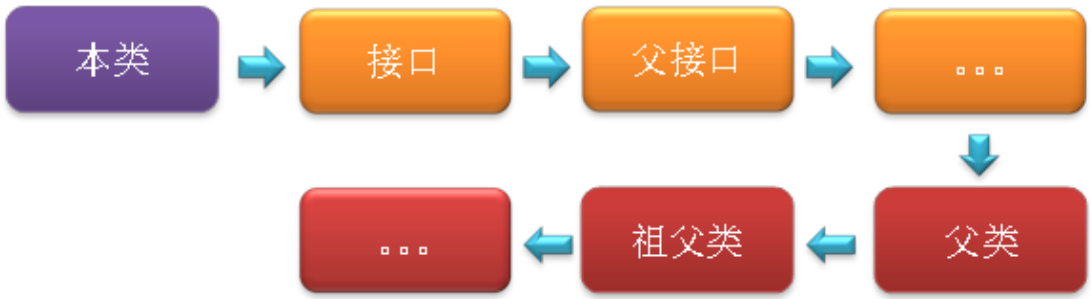
> **符号引用**：以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。
> **直接引用**：直接引用可以是直接指向目标的指针、相对偏移量或是一个能直接定位到目标的句柄。

前面说解析阶段可能开始于初始化之前，也可能在初始化之后开始，虚拟机会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析（初始化之前），还是等到一个符号引用将要被使用前才去解析它（初始化之后）。

对同一个符号引用进行多次解析请求时很常见的事情，虚拟机实现可能会对第一次解析的结果进行缓存（在运行时常量池中记录直接引用，并把常量标示为已解析状态），从而避免解析动作重复进行。

解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行，分别对应于常量池中的**CONSTANT_Class_info**、**CONSTANT_Fieldref_info**、**CONSTANT_Methodref_info**、**CONSTANT_InterfaceMethodref_info**四种常量类型。

1. **类或接口的解析**：判断所要转化成的直接引用是对数组类型，还是普通的对象类型的引用，从而进行不同的解析。
2. **字段解析：**对字段进行解析时，会先在本类中查找是否包含有简单名称和字段描述符都与目标相匹配的字段，如果有，则查找结束；如果没有，则会按照继承关系从上往下递归搜索该类所实现的各个接口和它们的父接口，还没有，则按照继承关系从上往下递归搜索其父类，直至查找结束，查找流程如下图所示：



这里我们便可以分析如下：static变量发生在静态解析阶段，也即是初始化之前，此时已经将字段的符号引用转化为了内存引用，也便将它与对应的类关联在了一起，由于在子类中没有查找到与m相匹配的字段，那么m便不会与子类关联在一起，因此并不会触发子类的初始化。

> 理论上是按照上述顺序进行搜索解析，但在实际应用中，虚拟机的编译器实现可能要比上述规范要求的更严格一些。如果有一个同名字段同时出现在该类的接口和父类中，或同时在自己或父类的接口中出现，编译器可能会拒绝编译。
>
> 编译时会报出如下错误：
> StaticTest.java:24: 对 m 的引用不明确，Father 中的 变量 m 和 Super 中的变量m都匹配
> System.out.println(Child.m);

1. **类方法解析：**对类方法的解析与对字段解析的搜索步骤差不多，只是多了判断该方法所处的是类还是接口的步骤，而且对类方法的匹配搜索，是先搜索父类，再搜索接口。
2. **接口方法解析：**与类方法解析步骤类似，知识接口不会有父类，因此，只递归向上搜索父接口就行了。

初始化

初始化是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序通过自定义的加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码（或者说是字节码）。

在准备阶段，类变量已经被赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序指定的主观计划去初始化类变量和其他资源，或者可以从另一个角度来表达：初始化阶段是执行类构造器()方法的过程。

- （）方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句中可以赋值，但是不能访问。
- （）方法与实例构造器（）方法（类的构造函数）不同，它不需要显式地调用父类构造器，虚拟机会保证在子类的（）方法执行之前，父类的（）方法已经执行完毕。因此，在虚拟机中第一个被执行的（）方法的类肯定是

java.lang.Object。

- （）方法对于类或接口来说并不是必须的，如果一个类中没有静态语句块，也没有对类变量的赋值操作，那么编译器可以不为这个类生成（）方法。
- 接口中不能使用静态语句块，但仍然有类变量（final static）初始化的赋值操作，因此接口与类一样会生成（）方法。但是接口鱼类不同的是：执行接口的（）方法不需要先执行父接口的（）方法，只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的（）方法。
- 虚拟机会保证一个类的（）方法在多线程环境中被正确地加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的（）方法，其他线程都需要阻塞等待，直到活动线程执行（）方法完毕。如果在一个类的（）方法中有耗时很长的操作，那就可能造成多个线程阻塞，在实际应用中这种阻塞往往是很隐蔽的。

总结

整个类加载过程中，除了在加载阶段用户应用程序可以自定义类加载器参与之外，其余所有的动作完全由虚拟机主导和控制。到了初始化才开始执行类中定义的Java程序代码（亦及字节码），但这里的执行代码只是个开端，它仅限于（）方法。类加载过程中主要是将Class文件（准确地讲，应该是类的二进制字节流）加载到虚拟机内存中，真正执行字节码的操作，在加载完成后才真正开始。

类加载器

类加载器虽然只用于实现类的加载动作，但它在Java程序中起到的作用却远远不限于类的加载阶段。对于任意一个类，都需要由它的类加载器和这个类本身一同确定其在就Java虚拟机中的唯一性，也就是说，即使两个类来源于同一个Class文件，只要加载它们的类加载器不同，那这两个类就必定不相等。这里的“相等”包括了代表类的Class对象的equals（）、isAssignableFrom（）、isInstance（）等方法的返回结果，也包括了使用instanceof关键字对对象所属关系的判定结果。

站在Java虚拟机的角度来讲，只存在两种不同的类加载器：

- **启动类加载器 Bootstrap ClassLoader**：它使用C++实现（这里仅限于Hotspot，也就是JDK1.5之后默认的虚拟机，有很多其他的虚拟机是用Java语言实现的），是虚拟机自身的一部分。
- **所有其他的类加载器**：这些类加载器都由Java语言实现，独立于虚拟机之外，并且全部继承自抽象类java.lang.ClassLoader，这些类加载器需要由启动类加载器加载到内存中之后才能去加载其他的类。

站在Java开发人员的角度来看，类加载器可以大致划分为以下三类：

- **启动类加载器：**Bootstrap ClassLoader，跟上面相同。它负责加载存放在**JDK\jre\lib**(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库（如rt.jar，所有的java.*开头的类均被Bootstrap ClassLoader加载）。启动类加载器是无法被Java程序直接引用的。
- **扩展类加载器：**Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载**JDK\jre\lib\ext**目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.*开头的类），开发者可以直接使用扩展类加载器。
- **应用程序类加载器**：Application ClassLoader，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（**ClassPath**）所指定的类，开发者可以直接使用该类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

应用程序都是由这三种类加载器互相配合进行加载的，如果有必要，我们还可以加入自定义的类加载器。因为JVM自带的ClassLoader只是懂得从本地文件系统加载标准的java class文件，因此如果编写了自己的ClassLoader，便可以做到如下几点：

1）在执行非置信代码之前，自动验证数字签名。

2）动态地创建符合用户特定需要的定制化构建类。

3）从特定的场所取得Java class，例如数据库中和网络中。

事实上当使用Applet的时候，就用到了特定的ClassLoader，因为这时需要从网络上加载java class，并且要检查相关的安全信息，应用服务器也大都使用了自定义的ClassLoader技术。

这种层次关系称为类加载器的双亲委派模型。我们把每一层上面的类加载器叫做当前层类加载器的父加载器，当然，它们之间的父子关系并不是通过继承关系来实现的，而是使用组合关系来复用父加载器中的代码。

双亲委派模型的工作流程是：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加载器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当父加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该类。

使用双亲委派模型来组织类加载器之间的关系，有一个很明显的好处，就是Java类随着它的类加载器（说白了，就是它所在的目录）一起具备了**一种带有优先级的层次关系，这对于保证Java程序的稳定运作很重要**。例如，类java.lang.Object类存放在JDK\jre\lib下的rt.jar之中，因此无论是哪个类加载器要加载此类，最终都会委派给启动类加载器进行加载，这边保证了Object类在程序中的各种类加载器中都是同一个类。

> 递归调用来实现，不断轮训父加载器有没有该类。

如果没用双亲委派模型的话，那么各个类加载器自行去加载的话，如果用户自己编写了一个称为java.lang.object的类，并放在程序的ClassPath中，那系统中将会出现多个不同的Object类，Java类型体系中最基础的行为也就无法保证，应用程序也就会变得一片混乱。

反射和代理、异常、Java8相关、序列化

反射

通过反射，我们可以在运行时获得程序或程序集中每一个类型的成员和成员的信息。

程序中一般的对象的类型都是在编译期就确定下来的，而Java反射机制可以动态地创建对象并调用其属性，这样的对象的类型在编译期是未知的。所以我们可以通过反射机制直接创建对象，即使这个对象的类型在编译期是未知的。

反射的核心是JVM在运行时才动态加载类或调用方法/访问属性，它不需要事先（写代码的时候或编译期）知道运行对象是谁。

Java反射框架主要提供以下功能：

1. 在运行时判断任意一个对象所属的类；
2. 在运行时构造任意一个类的对象；
3. 在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用private方法）；
4. 在运行时调用任意一个对象的方法

> 重点：是运行时而不是编译时

反射的主要用途

当我们在使用IDE(如Eclipse，IDEA)时，当我们输入一个对象或类并想调用它的属性或方法时，一按点号，编译器就会自动列出它的属性或方法，这里就会用到反射。

Java反射机制可以动态地获取类的结构，动态地调用对象的方法，是java语言一个动态化的机制。

反射最重要的用途就是**开发各种通用框架**。

很多框架（比如Spring）都是配置化的（比如通过XML文件配置JavaBean,Action之类的），为了保证框架的通用性，它们可能需要根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射——运行时动态加载需要加载的对象。

反射的基本运用

上面我们提到了反射可以用于判断任意对象所属的类，获得Class对象，构造任意一个对象以及调用一个对象。这里我们介绍一下基本反射功能的实现(反射相关的类一般都在java.lang.relfect包里)。

1、获得Class对象

方法有三种
（1）使用Class类的forName静态方法:

```
 public static Class<?> forName(String className)

\\在JDBC开发中常用此方法加载数据库驱动：

 Class.forName(driver);
```

（2）直接获取某一个对象的class，比如：

```
Class<?> klass = int.class;
Class<?> classInt = Integer.TYPE;
```

(3)调用某个对象的getClass()方法,比如:

```
StringBuilder str = new StringBuilder("123");
Class<?> klass = str.getClass();
```

2、判断是否为某个类的实例

一般地，我们用instanceof关键字来判断是否为某个类的实例。同时我们也可以借助反射中Class对象的isInstance()方法来判断是否为某个类的实例，它是一个Native方法：

```
public native boolean isInstance(Object obj);

Class<?> klass = str.getClass();
klass.isInstance(str);
```

3、创建实例
通过反射来生成对象主要有两种方式。
（1）使用Class对象的newInstance()方法来创建Class对象对应类的实例。

```
Class<?> c = String.class;
Object str = c.newInstance();
```

(2)先通过Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance()方法来创建实例。这种方法可以用指定的构造器构造类的实例。

```
//获取String所对应的Class对象
Class<?> c = String.class;
//获取String类带一个String参数的构造器
Constructor constructor = c.getConstructor(String.class);
//根据构造器创建实例
Object obj = constructor.newInstance("23333");
System.out.println(obj);
```

4、获取方法
获取某个Class对象的方法集合，主要有以下几个方法：
getDeclaredMethods()方法返回类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。

```
public Method[] getDeclaredMethods() throws SecurityException
```

getMethods()方法返回某个类的所有公用（public）方法，包括其继承类的公用方法。

```
public Method[] getMethods() throws SecurityException
```

getMethod方法返回一个特定的方法，其中第一个参数为方法名称，后面的参数为方法的参数对应Class的对象

```
public Method getMethod(String name, Class<?>... parameterTypes)
```

5、获取构造器信息
获取类构造器的用法与上述获取方法的用法类似。主要是通过Class类的getConstructor方法得到Constructor类的一个实例，而Constructor类有一个newInstance方法可以创建一个对象实例:

```
public T newInstance(Object ... initargs)
```

6、获取类的成员变量（字段）信息

```
getFiled：访问公有的成员变量
getDeclaredField：所有已声明的成员变量。但不能得到其父类的成员变量
getFileds和getDeclaredFields用法同上（参照Method）
```

7、调用方法

当我们从类中获取了一个方法后，我们就可以用**invoke()**方法来调用这个方法。invoke方法的原型为:

```
public Object invoke(Object obj, Object... args)
    throws IllegalAccessException, IllegalArgumentException,
       InvocationTargetException
```

```
public class test1 {
    public static void main(String[] args) throws IllegalAccessException, InstantiationExcepti
on, NoSuchMethodException, InvocationTargetException {
        Class<?> klass = methodClass.class;
        //创建methodClass的实例
        Object obj = klass.newInstance();
        //获取methodClass类的add方法
        Method method = klass.getMethod("add",int.class,int.class);
        //调用method对应的方法 => add(1,4)
        Object result = method.invoke(obj,1,4);
        System.out.println(result);
    }
}
```

反射的一些注意事项

由于反射会额外消耗一定的系统资源，因此如果不需要动态地创建一个对象，那么就不需要用反射。
另外，反射调用方法时可以忽略权限检查，因此可能会破坏封装性而导致安全问题。

代理

1. 静态代理：由程序员创建或特定工具自动生成源代码，再对其编译。在程序运行前，代理类的.class文件就已经存在了。
2. 动态代理：在程序运行时，由Java反射机制动态生成字节码。

- 静态代理

```
public interface Count {
  public void queryCount();
}
public class CountImpl implements Count {
  public void queryCount() {
    System.out.println("查看账户方法...");
  }
}
//代理类
public class CountProxy implements Count {
  private CountImpl countImpl;
  public CountProxy(CountImpl countImpl) {
    this.countImpl = countImpl;
  }
  @Override
  public void queryCount() {
    System.out.println("事务处理之前");
    countImpl.queryCount(); // 调用委托类的方法;
    System.out.println("事务处理之后");
  }
}
//测试类
public class TestCount {
  public static void main(String[] args) {
    CountImpl countImpl = new CountImpl();
    CountProxy countProxy = new CountProxy(countImpl);
    countProxy.queryCount();
  }
}
```

观察代码可以发现每一个代理类只能为一个接口服务，这样一来程序开发中必然会产生过多的代理，而且，所有的代理操作除了调用的方法不一样之外，其他的操作都一样，则此时肯定是重复代码。解决这一问题最好的做法是可以通过一个代理类完成全部的代理功能，那么此时就必须使用动态代理完成。

- 动态代理

动态代理类的字节码在程序运行时由Java反射机制动态生成，无需程序员手工编写它的源代码。动态代理类不仅简化了编程工作，而且提高了软件系统的可扩展性，因为Java 反射机制可以生成任意类型的动态代理类。

java.lang.reflect 包中的**Proxy类**和**InvocationHandler接口**提供了生成动态代理类的能力。

InvocationHandler接口：

```
public interface InvocationHandler {
public Object invoke(Object proxy,Method method,Object[] args) throws Throwable;
}
参数说明：
Object proxy：指被代理的对象。
Method method：要调用的方法
Object[] args：方法调用时所需要的参数
可以将InvocationHandler接口的子类想象成一个代理的最终操作类，替换掉ProxySubject。
```

Proxy类：

```
Proxy类是专门完成代理的操作类，可以通过此类为一个或多个接口动态地生成实现类，
此类提供了如下的操作方法：
public static Object newProxyInstance(ClassLoader loader,
Class<?>[] interfaces, InvocationHandler h)
throws IllegalArgumentException

参数说明：
ClassLoader loader：类加载器
Class<?>[] interfaces：得到全部的接口
InvocationHandler h：得到InvocationHandler接口的子类实例
```

```
interface Subject {
  public String say(String name, int age);
}
class RealSubject implements Subject {
  @Override
  public String say(String name, int age) {
    return name + " " + age;
  }
}
//JDK动态代理类
class MyInvocationHandler implements InvocationHandler {
  private Object target = null;
  //绑定委托对象并返回一个代理类
  public Object bind(Object target) {
    this. target = target;
    return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                target.getClass().getInterfaces(), this); //要绑定接口（cglib弥补了这一点）
  }
  @Override
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    System.out.println(“before method!”);
    Object temp = method.invoke(target, args);
    System.out.println(“after method!”);
    return temp;
  }
}
class hello {
  public static void main(String[] args) {
    MyInvocationHandler demo = new MyInvocationHandler();
    Subject sub = (Subject) demo.bind(new RealSubject());
    String info = sub.say("Rollen", 20);
    System.out.println(info);
  }
}
```

设计模式（常用的，jdk中有的）

在项目中的应用。

Web相关

```
Servlet、cookie/session、Spring<AOP/IPC/MVC/事务/动态代理>
MyBatis、Tomcat、Hibernate等、SSH
```

其他

线程和进程有什么区别?

线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。别把它和栈内存搞混，每个线程都拥有单独的栈内存用来存储本地数据。

Thread 类中的start() 和 run() 方法有什么区别?

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

Java中Runnable和Callable有什么不同?

Runnable和Callable都代表那些要在不同的线程中执行的任务。Runnable从JDK1.0开始就有了，Callable是在JDK1.5增加的。它们的主要区别是Callable的 call() 方法可以返回值和抛出异常，而Runnable的run()方法没有这些功能。Callable可以返回装载有计算结果的Future对象。

Java中CyclicBarrier 和 CountDownLatch有什么不同?

CyclicBarrier 和 CountDownLatch 都可以用来让一组线程等待其它线程。与 CyclicBarrier 不同的是，CountdownLatch 不能重新使用。

什么是线程安全？Vector是一个线程安全类吗?

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分成两组，线程安全和非线程安全的。Vector 是用同步方法来实现线程安全的, 而和它相似的ArrayList不是线程安全的。

Java中什么是竞态条件?

竞态条件会导致程序在并发情况下出现一些bugs。多线程对一些资源的竞争的时候就会产生竞态条件，如果首先要执行的程序竞争失败排到后面执行了，那么整个程序就会出现一些不确定的bugs。这种bugs很难发现而且会重复出现，因为线程间的随机竞争。

Java中如何停止一个线程?

Java提供了很丰富的API但没有为停止线程提供API。JDK 1.0本来有一些像stop(), suspend() 和 resume()的控制方法但是由于潜在的死锁威胁因此在后续的JDK版本中他们被弃用了，之后Java API的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。当run() 或者 call() 方法执行完的时候线程会自动结束,如果要手动结束一个线程，你可以用volatile 布尔变量

来退出run()方法的循环或者是取消任务来中断线程。

为什么wait, notify 和 notifyAll这些方法不在thread类里面?

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

什么是FutureTask?

在Java并发程序中FutureTask表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。只有当运算完成的时候结果才能取回，如果运算尚未完成get方法将会阻塞。一个FutureTask对象可以对调用了Callable和Runnable的对象进行包装，由于FutureTask也是调用了Runnable接口所以它可以提交给Executor来执行。

Java中堆和栈有什么不同?

为什么把这个问题归类在多线程和并发面试题里？因为栈是一块和线程紧密相关的内存区域。每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的。而堆是所有线程共享的一片公用内存区域。对象都在堆里创建，为了提升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时volatile 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

如何写代码来解决生产者消费者问题?

在现实中你解决的许多线程问题都属于生产者消费者模型，就是一个线程生产任务供其它线程进行消费，你必须知道怎么进行线程间通信来解决这个问题。比较低级的办法是用wait和notify来解决这个问题，比较赞的办法是用Semaphore 或者BlockingQueue来实现生产者消费者模型，这篇教程有实现它。

怎么检测一个线程是否拥有锁?

在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁。

Java中synchronized 和 ReentrantLock 有什么不同?

Java在过去很长一段时间只能通过synchronized关键字来实现互斥，它有一些缺点。比如你不能扩展锁之外的方法或者块边界，尝试获取锁时不能中途取消等。Java 5 通过Lock接口提供了更复杂的控制来解决这些问题。 ReentrantLock 类实现了Lock，它拥有与 synchronized 相同的并发性和内存语义且它还具有可扩展性。

有三个线程T1，T2，T3，怎么确保它们按顺序执行?

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

Thread类中的yield方法有什么作用?

Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。

Java中ConcurrentHashMap的并发度是什么?

ConcurrentHashMap把实际map划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16，这样在多线程情况下就能避免争用。

如果你提交任务时，线程池队列已满。会时发会生什么?

事实上如果一个任务不能被调度执行那么ThreadPoolExecutor's submit()方法将会抛出一个RejectedExecutionException异常。

Java线程池中submit() 和 execute()方法有什么区别?

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

什么是阻塞式方法?

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，ServerSocket的accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

Swing是线程安全的吗？ 为什么?

你可以很肯定的给出回答，Swing不是线程安全的，但是你应该解释这么回答的原因即便面试官没有问你为什么。
当我们说swing不是线程安全的常常提到它的组件，这些组件不能在多线程中进行修改，所有对GUI组件的更新都要在AWT线程中完成，而Swing提供了同步和异步两种回调方法来进行更新。

Swing API中那些方法是线程安全的?

这个问题又提到了swing和线程安全，虽然组件不是线程安全的但是有一些方法是可以被多线程安全调用的，比如repaint(), revalidate()。 JTextComponent的setText()方法和JTextArea的insert() 和 append() 方法也是线程安全的。

volatile 变量和 atomic 变量有什么不同?

这是个有趣的问题。首先，volatile 变量和 atomic 变量看起来很像，但功能却不一样。Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

写出3条你遵循的多线程最佳实践

这种问题我最喜欢了，我相信你在写并发代码来提升性能的时候也会遵循某些最佳实践。以下三条最佳实践我觉得大多数Java程序员都应该遵循：

- 给你的线程起个有意义的名字。 这样可以方便找bug或追踪。OrderProcessor, QuoteProcessor or TradeProcessor 这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。

- 避免锁定和缩小同步的范围 锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。
- 多用同步类少用wait 和 notify 首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。
- 多用并发集合少用同步集合 这是另外一个容易遵循且受益巨大的最佳实践，并发集合比同步集合的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap

如何强制启动一个线程?

这个问题就像是如何强制进行Java垃圾回收，目前还没有觉得方法，虽然你可以使用System.gc()来进行垃圾回收，但是不保证能成功。在Java里面没有办法强制启动一个线程，它是被线程调度器控制着且Java没有公布相关的API。

Java多线程中调用wait() 和 sleep()方法有什么不同?

Java程序中wait 和 sleep都会造成某种形式的暂停，它们可以满足不同的需要。wait()方法用于线程间通信，如果等待条件为真且其它线程被唤醒时它会释放锁，而sleep()方法仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁。

ArrayList是如何实现的，和LinkedList的区别？ArrayList如何实现扩容。