## Java基础

###### Java安装

Oracle官网下载安装

###### Java环境变量

JAVA\_HOME Classpath

为啥需要配置环境变量?

###### 版本

Java8 最重要的新特性

Stream,函数式编程

## Maven基础

###### 安装

<http://maven.apache.org/download.cgi>

###### 环境变量

加入PATH中

###### 关键配置

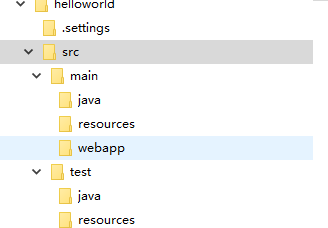
settings.xml

pom.xml

搭建私库

<https://www.cnblogs.com/cac2020/p/9796168.html>





mian:该目录下存放的是该项目的主要代码文件，下面有三个文件夹，分别用来存放：

Java：该目录用来存放Java的源代码文件

resources：该目录主要用来存放项目的一些配置文件（比如spring的xml配置文件）。

webapp：该目录相当于web项目下webcontent的目录，用来存放js、jsp、web.xml等文件.

test:该目录用来存放项目的测试文件

java：该目录主要存放该项目的测试所用的Java源代码。

resources：该目录存放测试使用的资源文件。

## Git基础

###### 安装

下载安装

###### 常用操作

git init

git clone

git add

git commit 本地仓库

git revert 撤回到pull下来的状态

git pull 从远程拉取

git push 把本地的代码push到远程

git log

git checkout -b 从远程拉取分支，本地新建一个分支

git stash

git clone 🡪 git checkout -b 🡪 coding 🡪 git add 🡪 git commit 🡪 git push

<https://www.liaoxuefeng.com/wiki/896043488029600>

###### IDEA

相关配置， maven

常用插件 git，lombok

## JAVA笔记

每个Java程序最少都会有一个类和一个mian()。main()是程序的起点。

Java中所有东西都会属于某个类。.java是源文件，.class是类文件，真正被执行的是类。

类：由变量和方法构成。变量必须有类型和名称。对象是类的实例。

## Java 基础类型

本类型并不具有对象的性质，为了让基本类型也具有对象的特征，就出现了包装类型。如我们在使用集合类型Collection时就一定要使用包装类型而非基本类型。

基本类型　包装器类型

boolean Boolean

char Character

int Integer

byte Byte

short Short

long Long

float Float

double Double

Java为每种基本数据类型都提供了对应的包装器类型，从Java SE5开始就提供了自动装箱的特性。

装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的 xxxValue方法实现的。

不同之处：

1. 声明方式不同：

基本类型不使用new关键字，而包装类型需要使用new关键字来在堆中分配存储空间；

2. 存储方式及位置不同：

基本类型是直接将变量值存储在栈中，而包装类型是将对象放在堆中，然后通过引用来使用；

3. 初始值不同：

基本类型的初始值如int为0，boolean为false，而包装类型的初始值为null；

4. 使用方式不同：

基本类型直接赋值直接使用就好，而包装类型在集合如Collection、Map时会使用到。

## 变量

局部变量：定义在方法中的变量都是局部变量(main方法也是方法，所以定义在main方法中的变量也是局部变量)。

全局变量: 全局变量都是定在类中,分为静态和非静态。静态变量在方法区中，非静态在堆中。

变量命名只能以字母，\_或$开头，不能以数字开头。

数组也是对象，不是基本数据类型。

类的基本数据类型变量是有默认值的，非基本数据类型是引用，因此默认值是null。

在函数中的局部变量是没有默认值的，不论是否是基本数据类型。

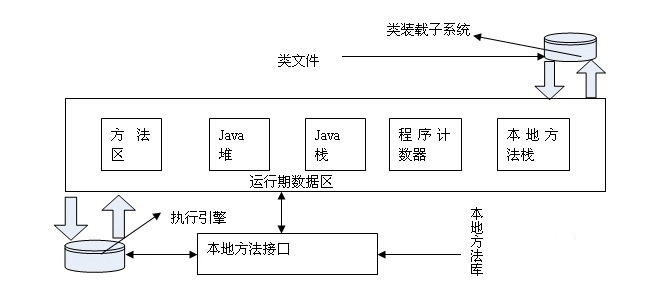
注意如果将对象的引用直接用新对象赋值，那么这个引用就会指向新的对象

== 比较，对于基本数据类型，就是用来比较值的。对于非基本数据类型，是比较是否指向同一个对象。

Integer a = 6;  
Integer b = 6;  
Integer c = 200;  
Integer d = 200;  
  
System.***out***.println(a==b ); true  
System.***out***.println(c==d ); false

Integer有缓存，使用equals比较

## 堆和栈



线程共享区：方法区、堆

线程私有区：虚拟机栈、本地方法栈、程序计数器

JVM内存的划分有五片：

1. 寄存器； 当前线程所执行的字节码的行号指示器

2. 本地方法区：本地方法栈是为虚拟机执行native方法的

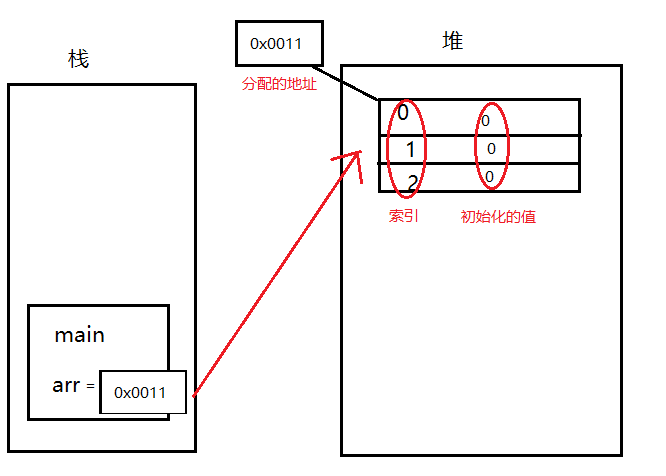
3. 方法区； 存储被虚拟机加载的类信息，常量，静态常量，静态方法等，常量池，用来存储编译期间生成的字面量和符号引用

4. 栈内存:

5. 堆内存:

栈内存:栈内存首先是一片内存区域，存储的都是局部变量，凡是定义在方法中的都是局部变量（方法外的是全局变量），方法先进栈，然后再定义变量

堆内存:存储的是数组和对象（其实数组就是对象），凡是new建立的都是在堆中，堆中存放的都是实体（对象），



如果当int [] arr=null;

  arr不做任何指向，null的作用就是取消引用数据类型的指向

## String

**StringBuilder 非线程安全 用在函数中**

StringBuffer 线程安全，全局变量需要使用stringbuffer

## 面向对象

封装，继承，多态

 **面向对象的三大特性**：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。

 **多态的定义**：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。

 **实现多态的技术称为**：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法

用子类实例化父类，在运行期间才会决定调用谁的方法，因此会首先调用实例化的方法，找不到在调用父类方法。

如果函数f是final(无法继承)、static(根据引用类型去获取方法)

如果超类没有无参构造函数，而子类没有显示的调用超类的其他构造函数（子类构造函数会默认的调用父类的无参构造函数），则java编译器会报错。

super(..)调用父类构造函数，必须放在子类构造函数第一行

重载和覆盖不一样，重载和覆盖没有任何关系。重载可以改变参数类型和数量、返回值类型、权限,而覆盖参数和返回值，权限都要和父类的一致。

## 常用数据结构

List Map Set 数组， queue，stack

List: add(object), add(i，object),remove(),get()

LinkedList 插入块，遍历慢

ArrayList 遍历快，修改慢

Map: put(key,value),get(key)

HashMap 无序

LinkedHashMap 有序

Stack：

先进后出 push(obj),pop()

Queue：

先进先出

ArrayBlockingQueue put(obj),get()

Set: add, get,remove 没有value的map

HashSet 无序

TreeSet 有序

## 泛型

泛型，即“参数化类型”。

顾名思义，就是将类型由原来的具体的类型参数化

操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

也就是说Java中的泛型，只在编译阶段有效。

泛型类型在逻辑上看以看成是多个不同的类型，实际上都是相同的基本类型。

###### 泛型类

泛型类型用于类的定义中，被称为泛型类。通过泛型可以完成对一组类的操作对外开放相同的接口。最典型的就是各种容器类，如：List、Set、Map。

###### 泛型接口

泛型接口与泛型类的定义及使用基本相同。

###### 泛型方法

public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,

IllegalAccessException{

T instance = tClass.newInstance();

return instance;

}

###### 泛型上下边界

为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

为泛型添加上边界，即传入的类型实参必须是指定类型的子类型

* <? extends T>：是指 “上界通配符（Upper Bounds Wildcards）”

上界<? extends T>不能往里存，只能往外取

* <? super T>：是指 “下界通配符（Lower Bounds Wildcards）”

下界<? super T>不影响往里存，但往外取只能放在Object对象里

频繁往外读取内容的，适合用上界Extends。

经常往里插入的，适合用下界Super。

## Stream

Stream（流）是一个来自数据源的元素队列并支持聚合操作

* **数据源** 流的来源。 可以是集合，数组，I/O channel， 产生器generator 等。
* **聚合操作** 类似SQL语句一样的操作， 比如filter, map, reduce, find, match, sorted等。
* 中间操作都会返回流对象本身。 这样多个操作可以串联成一个管道， 如同流式风格（fluent style）。 这样做可以对操作进行优化， 比如延迟执行(laziness)和短路( short-circuiting)。

在 Java 8 中, 集合接口有两个方法来生成流：

* **stream()** − 为集合创建串行流。 Stream.of()
* **parallelStream()** − 为集合创建并行流。

forEach 来迭代流中的每个数据

map 方法用于映射每个元素到对应的结果

System.***out***.println(list1.stream().map(a -> String.*valueOf*(a))  
 .collect(Collectors.*joining*(**","**))  
);

filter 方法用于通过设置的条件过滤出元素

limit 方法用于获取指定数量的流。

Sorted 方法用于对流进行排序。

Collectors 类实现了很多归约操作，例如将流转换成集合和聚合元素

## 函数式编程

函数式接口(Functional Interface)

只定义一个抽象方法的接口，以函数式接口为参数的方法就可以使用Lambda表达式。

Java8内置函数式接口

Predicate：定义了 boolean test(T t) 抽象方法，需要表示一个涉及类型T的布尔表达式时可以使用

Consumer：定义了 void accept(T t) 抽象方法，需要访问某对象并对其进行某些操作时可以使用

Function：定义了 R apply(T t) 抽象方法，它接受一个 泛型T的对象，并返回一个泛型R的对象。

Supplier：定义了 T get() 抽象方法，不接收参数返回Lambda表达式的值

BiFunction：定义 R apply(T t, U u) 抽象方法，接收t和u参数，返回R对象，如果需要两个对象中的某些值来组装成另一个对象

BiPredicate：定义了 boolean test(T t, U u) 抽象方法，接收 t 和 u参数，返回比较的接口，需要两个对象做比较可以使用

BiConsumer：定义了 void accept(T t, U u) 抽象方法

函数式接口可以被隐式转换为 lambda 表达式。

@FunctionalInterface

interface GreetingService

{

void sayMessage(String message);

}

## 多线程

多线程只有一个目的，那就是更好的利用cpu的资源。

多线程：指的是这个程序（一个进程）运行时产生了不止一个线程

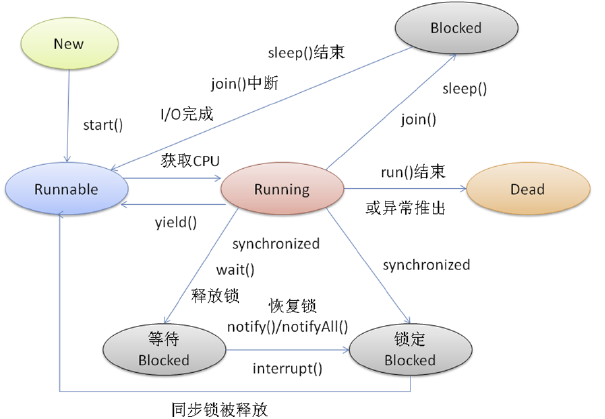
并行与并发：

并行：多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时。

并发：通过cpu调度算法，让用户看上去同时执行，实际上从cpu操作层面不是真正的同时。

线程安全：指在并发的情况之下，该代码经过多线程使用，线程的调度顺序不影响任何结果。

## 线程的状态



线程在Running的过程中可能会遇到阻塞(Blocked)情况

1. 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
2. 调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool )，释放同步锁使线程回到可运行状态（Runnable）
3. 对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool ),同步锁被释放进入可运行状态(Runnable)。

此外，在runnable状态的线程是处于被调度的线程，此时的调度顺序是不一定的。Thread类中的yield方法可以让一个running状态的线程转入runnable。

## 基本线程类

基本线程类指的是Thread类，Runnable接口，Callable接口。

1、继承Thread类实现多线程

public class MyThread extends Thread {

　　public void run() {

　　 System.out.println("MyThread.run()");

　　}

}

在合适的地方启动线程如下：

MyThread myThread1 = new MyThread();

myThread1.start(); //启动线程

myThread1.run; //只是执行run方法，不会新建一个线程

2、实现Runnable接口方式实现多线程

如果自己的类已经extends另一个类，就无法直接extends Thread，此时，必须实现一个Runnable接口，如下：

public class MyThread extends OtherClass implements Runnable {

　　public void run() {

　　 System.out.println("MyThread.run()");

　　}

}

为了启动MyThread，需要首先实例化一个Thread，并传入自己的MyThread实例：

MyThread myThread = new MyThread();

Thread thread = new Thread(myThread);

thread.start();

3.实现Callable接口，重写call方法（有返回值）

自定义类实现Callable接口时，必须指定泛型，该泛型即返回值的类型。Callable 依赖 FutureTask 类获取返回结果。

public static void main(String[] args) throws Exception {

MyThread mt = new MyThread();

FutureTask<Integer> result = new FutureTask<Integer>(mt);

new Thread(result).start();

// 获取运算结果是同步过程，即 call 方法执行完成，才能获取结果

Integer sum = result.get();

System.out.println(sum);

}

}

class MyThread implements Callable<Integer> {

@Override

public Integer call() throws Exception {

int sum = 0;

for (int i = 1; i <= 100; i++) {

sum += i;

}

return sum;

}

}

4、使用ExecutorService线程池

Executor 负责现成的使用和调度的根接口

\* |--ExecutorService 线程池的主要接口

\* |--ThreadPoolExecutor 线程池的实现类

\* |--ScheduledExecutorService 接口，负责线程的调度

\* |--ScheduledThreadPoolExecutor (extends ThreadPoolExecutor implements ScheduledExecutorService)

\*

\*

\* Executors工具类

\* 提供了创建线程池的方法

## 线程池的类型及其应用场景

## newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

## newCachedThreadPool

创建一个可缓存的线程池。这种类型的线程池特点是：   
1).工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。   
2).如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

## newSingleThreadExecutor

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的 。

## newScheduleThreadPool

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于Timer。

### 几种类型的对比

FixedThreadPool

它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源.

CachedThreadPool  
特点是在线程池空闲时，即线程池中没有可运行任务时，它会释放工作线程，从而释放工作线程所占用的资源。但是，但当出现新任务时，又要创建一新的工作线程，又要一定的系统开销。并且，在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

# 线程池方法

**shutDown()**   
　　当线程池调用该方法时,线程池的状态则立刻变成SHUTDOWN状态。此时，则不能再往线程池中添加任何任务，否则将会抛出RejectedExecutionException异常。但是，此时线程池不会立刻退出，直到添加到线程池中的任务都已经处理完成，才会退出。   
**shutdownNow()**

线程池的状态立刻变成STOP状态，并**试图**停止所有正在执行的线程，不再处理还在池队列中等待的任务，当然，**它会返回那些未执行的任务**。   
它试图终止线程的方法是通过调用Thread.interrupt()方法来实现的，如果线程中没有sleep 、wait、Condition、定时锁等应用, interrupt()方法是无法中断当前的线程的。所以，ShutdownNow()并不代表线程池就一定立即就能退出，它可能必须要等待所有正在执行的任务都执行完成了才能退出。

**waitTermination**

接收人timeout和TimeUnit两个参数，用于设定超时时间及单位。当等待超过设定时间时，会监测ExecutorService是否已经关闭，若关闭则返回true，否则返回false。一般情况下会和shutdown方法组合使用。

**isShutDown**

当调用shutdown()或shutdownNow()方法后返回为true。

**isTerminated**

当调用shutdown()方法后，并且所有提交的任务完成后返回为true;

当调用shutdownNow()方法后，成功停止后返回为true;

## Callable,Future,Runnable

java.lang.Runnable它是一个接口，在它里面只声明了一个run()方法：

public interface Runnable {

public abstract void run();

}

由于run()方法返回值为void类型，所以在执行完任务之后无法返回任何结果。

Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()：

public interface Callable<V> {

/\*\*

\* Computes a result, or throws an exception if unable to do so.

\*

\* @return computed result

\* @throws Exception if unable to compute a result

\*/

V call() throws Exception;

}

可以看到，这是一个泛型接口，该接口声明了一个名称为call()的方法，同时这个方法可以有返回值V，也可以抛出异常。

一般情况下是配合ExecutorService来使用的，在ExecutorService接口中声明了若干个submit方法的重载版本：

<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

第一个方法：submit提交一个实现Callable接口的任务，并且返回封装了异步计算结果的Future。

第二个方法：submit提交一个实现Runnable接口的任务，并且指定了在调用Future的get方法时返回的result对象。

第三个方法：submit提交一个实现Runnable接口的任务，并且返回封装了异步计算结果的Future。

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，**该方法会阻塞直到任务返回结果。**

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;

}

* cancel方法用来取消任务，如果取消任务成功则返回true，如果取消任务失败则返回false。参数mayInterruptIfRunning表示是否允许取消正在执行却没有执行完毕的任务，如果设置true，则表示可以取消正在执行过程中的任务。如果任务还没有执行，则无论mayInterruptIfRunning为true还是false，肯定返回true。
* isCancelled方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。
* isDone方法表示任务是否已经完成，若任务完成，则返回true；
* get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；
* get(long timeout, TimeUnit unit)用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null。

也就是说Future提供了三种功能：

1）判断任务是否完成；

2）能够中断任务；

3）能够获取任务执行结果。

1、Callable规定的方法是call(),Runnable规定的方法是run()。

2、Callable的任务执行后可返回值，而Runnable的任务是不能返回值(是void)。

3、call方法可以抛出异常，run方法不可以。

4、运行Callable任务可以拿到一个Future对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

5、加入线程池运行，Runnable使用ExecutorService的execute方法，Callable使用submit方法。

## 线程常用方法

| **方法** | **含义** |
| --- | --- |
| sleep() | 暂停阻塞等待一段时间，时间过了就继续。 |
| wait() | 也是阻塞和等待，但是需要notify来唤醒。 |
| join() | 在一个线程中调用other.join(),将等待other执行完后才继续本线程 |
| notify()、notifyAll() | 唤醒线程 |
| yield() | 当前线程可转让cpu控制权，让别的就绪状态线程运行（切换），也会等待阻塞一段时间，但是时间不是由客户控制了 |
| interrupte() | 打断线程，代替过时方法stop() |

**常用方法**

Sleep：Thread类的静态方法，此线程暂停执行一段时间，而把执行机会让给其他线程，等到计时时间一到，此线程会自动苏醒。调用sleep()方法并不会释放锁。

Wait： object类的方法，用于线程间通信，必须放在同步控制方法和同步代码块中使用。这个方法会使当前拥有该对象锁的进程等待，线程会释放掉他所占用的锁，直到其他线程调用notify（）方法或者notifyAll（）时才醒来。

wait/notify必须存在于synchronized块中。

生产者伪代码：

count+1;

notify();

消费者伪代码：

while(count<=0)

wait()

count--

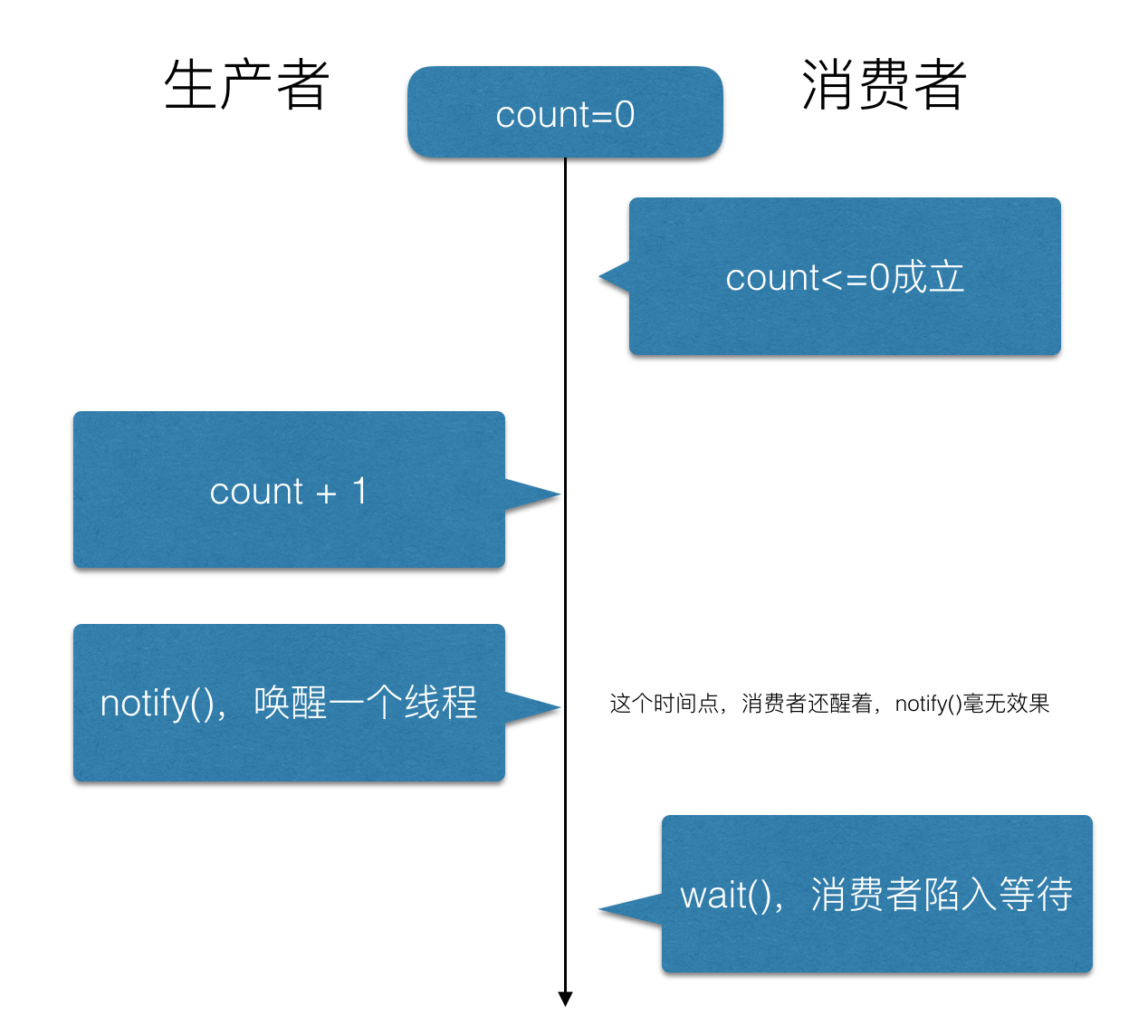
生产者是两个步骤：

1. count+1;
2. notify();

消费者也是两个步骤：

1. 检查count值；
2. 睡眠或者减一；

　　初始的时候count等于0，这个时候消费者检查count的值，发现count小于等于0的条件成立；就在这个时候，发生了上下文切换，生产者进来了，把两个步骤都执行完了，也就是发出了通知，准备唤醒一个线程。这个时候消费者刚决定睡觉，还没睡呢，所以这个上面唤醒通知就会被丢掉。



这就是所谓的lost wake up问题。(丢掉了唤醒线程的那条信息)

## 线程安全

**核心：共享变量的处理**

**Monitor**

首先就要明确monitor的概念，Java中的每个对象都有一个监视器,每个类也有一个监视器，**来监测并发代码的重入**。锁住了一个对象，就是获得对象相关联的监视器。

一个线程可以允许多次对同一对象上锁.对于每一个对象来说,java虚拟机维护一个计数器,记录对象被加了多少次锁,没被锁的对象的计数器是0,线程每加锁一次,计数器就加1,每释放一次,计数器就减1.当计数器跳到0的时候,锁就被完全释放了.

wait/notify必须存在于synchronized块中。并且，这三个关键字针对的是同一个监视器（某对象的监视器）。这意味着wait之后，其他线程可以进入同步块执行。

当某代码并不持有监视器的使用权时去wait或notify，会抛出java.lang.IllegalMonitorStateException。也包括在synchronized块中去调用另一个对象的wait/notify，因为不同对象的监视器不同，同样会抛出此异常。

**Synchronized**

对象锁

在多线程环境下，synchronized块中的方法获取了对象的monitor，如果实例相同，那么只有一个线程能执行该块内容。

用法：

1. 代码块

2. 直接用于非静态方法

**类锁**

类锁是用于类的静态方法或者一个类的class对象上的，类的对象实例可以有很多个，但是每个类只有一个class对象，所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。

用法：

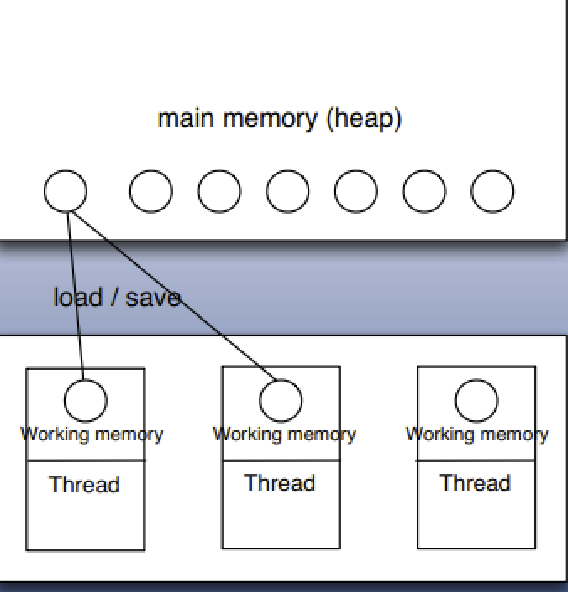
* 1. 代码块
  2. 静态方法

**类锁和对象锁是两个不一样的锁，控制着不同的区域，它们是互不干扰的**

**同步块内和同步方法都会强制刷新主内存**

##### volatile

多线程的内存模型：main memory（主存）、working memory（线程栈），在处理数据时，线程会把值从主存load到本地栈，完成操作后再save回去(volatile关键词的作用：每次针对该变量的操作都激发一次load and save)。



针对多线程使用的变量如果不是volatile或者final修饰的，很有可能产生不可预知的结果（另一个线程修改了这个值，但是之后在某线程看到的是修改之前的值）。其实道理上讲同一实例的同一属性本身只有一个副本。但是多线程是会缓存值的，本质上，volatile就是不去缓存，直接取值。在线程安全的情况下加volatile会牺牲性能。

**1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。**

**2）禁止进行指令重排序。**

**3）不保证原子性**

## 高级多线程控制类

### 1.ThreadLocal类

使用场景：ThreadLocal适用的场景是，**多个线程都需要使用一个变量，但这个变量的值不需要在各个线程间共享**，各个线程都只使用自己的这个变量的值。

ThreadLocal虽然它本身能够被多个线程共享使用，但并不是用来解决多线程环境下的共享变量问题，而是用来提供线程内部的共享变量, 可以保证各个线程之间的变量互相隔离、相互独立。

InheritableThreadLocal

ThreadLocal类是不能提供子线程访问父线程的本地变量的，而InheritableThreadLocal类则可以做到这个功能。

子线程优先获取自己线程的变量，获取不到去访问到父线程的变量，子线程无法修改父线程的变量。

ThreadLoal底层是个map，线程作为key，如果使用线程池，那么key就会一直存在，所以map中存的数据无法回收，在使用时候如果不需要threadlocal中的变量，需要使用remove.

### 2.原子类

保证操作的原子性。

**AtomicBoolean**

**AtomicInteger**

**AtomicLong**

### 3.Lock类

java.util.concurrent.locks.lock

lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。

tryLock()方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回true，如果获取失败（即锁已被其他线程获取），则返回false

tryLock(long time, TimeUnit unit)方法和tryLock()方法是类似的，只不过区别在于这个方法在拿不到锁时会等待一定的时间，在时间期限之内如果还拿不到锁，就返回false

lockInterruptibly()方法比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。

synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；**而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象**。

### 4.容器类

**CouncurrentHashMap**

HashMap在put的时候，插入的元素超过了容量（由负载因子决定）的范围就会触发扩容操作，就是rehash，这个会重新将原数组的内容重新hash到新的扩容数组中，在多线程的环境下，扩容同时其他的元素也在进行put操作，如果hash值相同，可能出现链表闭环，导致在get时会出现死循环，所以HashMap是线程不安全的。

CopyOnWriteArrayList

线程1通过Iterator在遍历集合A中的元素，同时线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常。

CopyOnWriteArrayList是使用CopyOnWrite（写时复制）技术解决了这个问题，这一般需要很大的内存开销。

## 线程流程控制

## CountDownLatch

## CyclicBarrier

栅栏类似于闭锁，它能阻塞一组线程直到某个事件的发生。栅栏与闭锁的关键区别在于，所有的线程必须同时到达栅栏位置，才能继续执行。闭锁用于等待事件，而栅栏用于等待其他线程。

CyclicBarrier可以使一定数量的线程反复地在栅栏位置处汇集。当线程到达栅栏位置时将调用await方法，这个方法将阻塞直到所有线程都到达栅栏位置。如果所有线程都到达栅栏位置，那么栅栏将打开，此时所有的线程都将被释放，而栅栏将被重置以便下次使用。

## Semaphore

**Semaphore用来控制同时访问某个特定资源的操作数量**，或者同时执行某个指定操作的数量。还可以用来实现某种资源池限制，或者对容器施加边界。

## 文件操作

创建文件

file.getParentFile().mkdirs();  
file.createNewFile();

修改文件

删除文件

file.exists()

file.delete();

文件分隔符 File.***pathSeparator***

文件 File

RandomAccessFile

可以读取文件内容，也可以写入，支持随机访问， 跳转到任意位置读写数据。

RandomAccessFile 对象包含一个记录指针，用以标识当前读写处的位置。  
下面方法记录指针：

* getFilePointer(): 返回指针当前位置
* seek(long pos): 将指针定位到 pos位置

## Socket

Socket

SocketServer

## 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法，对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。

反射使用的就是Class类中的方法。**所以先要获取到每一个字节码文件对应的Class类型的对象.**

**获取反射中的Class对象**  
在反射中，要获取一个类或调用一个类的方法，我们首先需要获取到该类的 Class 对象。  
在 Java API 中，获取 Class 类对象有三种方法：

1. 使用 Class.forName 静态方法。当知道某类的全路径名时，可以使用此方法获取 Class 类对象。

Class c1 = Class.forName(“java.lang.String”);

1. 直接通过 类名.class 的方式得到。任何一个类都有一个隐含的静态成员变量 class。

Class c2 = String.class;

1. 通过对象调用 getClass() 方法来获取，通常应用在：比如你传过来一个 Object类型的对象，而我不知道你具体是什么类，用这种方法。

Class.forName和getClass,.class的区别

forName() 方法发现某个累还没被加载，就会主动去加载这个类，在加载过程中，**该类的static 子句将会被执行。**

getClass,.class 会加载类，但是不会初始化这个类，**即不会调用static代码块。**

**通过反射创建类对象**  
通过反射创建类对象主要有两种方式：通过 Class 对象的 newInstance() 方法、通过 Constructor 对象的 newInstance() 方法。

1. 通过 Class 对象的 newInstance() 方法。  
   Class clz = Phone.class;  
   Phone phone = (Phone)clz.newInstance();
2. 通过 Constructor 对象的 newInstance() 方法

Class clz = Phone.class;

Constructor constructor = clz.getConstructor();

Phone phone= (Phone)constructor.newInstance();

通过 Constructor 对象创建类对象可以选择特定构造方法，而通过 Class 对象则只能使用**默认的无参数构造方法**。下面的代码就调用了一个有参数的构造方法进行了类对象的初始化。

Class clz = Phone.class;

Constructor constructor = clz.getConstructor(String.class, int.class);

Phone phone = (Phone)constructor.newInstance("66",6666);

getName()：获得类的完整名字。  
getFields()：获得类的public类型的属性。  
getDeclaredFields()：获得类的所有属性。包括private 声明的和继承类  
getMethods()：获得类的public类型的方法。  
getDeclaredMethods()：获得类的所有方法。包括private 声明的和继承类  
getMethod(String name, Class[] parameterTypes)：获得类的特定方法，name参数指定方法的名字，parameterTypes 参数指定方法的参数类型。  
getConstructors()：获得类的public类型的构造方法。  
getConstructor(Class[] parameterTypes)：获得类的特定构造方法，parameterTypes 参数指定构造方法的参数类型。  
newInstance()：通过类的不带参数的构造方法创建这个类的一个对象。

**new对象和反射得到对象的区别**

1. 在使用反射的时候，必须确保这个类已经加载并已经连接了。使用new的时候，这个类可以没有被加载，也可以已经被加载。
2. new关键字可以调用任何public构造方法，而反射只能调用无参构造方法。
3. new关键字是强类型的，效率相对较高。 反射是弱类型的，效率低。
4. 反射提供了一种更加灵活的方式创建对象，得到对象的信息。如Spring 中AOP等的使用，动态代理的使用，都是基于反射的。

## 代理

代理(Proxy)是一种设计模式,提供了对目标对象另外的访问方式;即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能.

代理模式的关键点是:代理对象与目标对象.代理对象是对目标对象的扩展,并会调用目标对象。

### 静态代理

静态代理在使用时,需要定义接口,被代理对象与代理对象一起实现相同的接口.

**静态代理总结:**  
1.可以做到在不修改目标对象的功能前提下,对目标功能扩展.  
2.缺点:

* 因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类,类太多.同时,一旦接口增加方法,目标对象与代理对象都要维护.

**动态代理**

**动态代理有以下特点:**  
1.代理对象,不需要实现接口,但是需要制定接口类型。  
2.代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)  
3.动态代理也叫做:JDK代理,接口代理

**JDK中生成代理对象的API**  
代理类所在包:java.lang.reflect.Proxy

JDK实现代理只需要使用newProxyInstance方法,但是该方法需要接收三个参数,完整的写法是:

static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler h )

注意该方法是在Proxy类中是静态方法,且接收的三个参数依次为:

* ClassLoader loader,:指定当前目标对象使用类加载器,获取加载器的方法是固定的
* Class<?>[] interfaces,:目标对象实现的接口的类型,使用泛型方式确认类型
* InvocationHandler h:事件处理,执行目标对象的方法时,会触发事件处理器的方法,会把当前执行目标对象的方法作为参数传入

### Cglib代理

静态代理和动态代理模式都是要求目标对象是实现一个接口的目标. 以目标对象子类的方式类实现代理,这种方法就叫做:Cglib代理.

Cglib代理,也叫作子类代理,它是在**内存中构建一个子类对象从而实现对目标对象功能的扩展.**

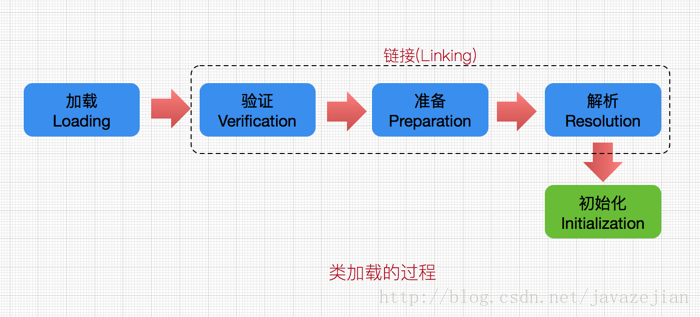
* JDK的动态代理有一个限制,就是使用动态代理的对象必须实现一个或多个接口,如果想代理没有实现接口的类,就可以使用Cglib实现.
* Cglib是一个强大的高性能的代码生成包,它可以在运行期扩展java类与实现java接口.它广泛的被许多AOP的框架使用,例如Spring AOP和synaop,为他们提供方法的interception(拦截)
* Cglib包的底层是通过使用一个小而块的字节码处理框架ASM来转换字节码并生成新的类.不鼓励直接使用ASM,因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉.

Cglib子类代理实现方法:  
1.需要引入cglib的jar文件.  
2.引入功能包后,就可以在内存中动态构建子类  
3.**代理的类不能为final,否则报错**  
4.目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.

## Classloader

**类加载的机制的层次结构**

每个编写的”.java”拓展名类文件都存储着需要执行的程序逻辑，这些”.java”文件经过Java编译器编译成拓展名为”.class”的文件，”.class”文件中保存着Java代码经转换后的虚拟机指令，当需要使用某个类时，虚拟机将会加载它的”.class”文件，并创建对应的class对象，将class文件加载到虚拟机的内存，这个过程称为类加载，这里我们需要了解一下类加载的过程，如下：



* 加载：类加载过程的一个阶段：通过一个类的完全限定查找此类字节码文件，并利用字节码文件创建一个Class对象
* 验证：目的在于确保Class文件的字节流中包含信息符合当前虚拟机要求，不会危害虚拟机自身安全。主要包括四种验证，文件格式验证，元数据验证，字节码验证，符号引用验证。
* 准备：为类变量(即static修饰的字段变量)分配内存并且设置该类变量的初始值即0(如static int i=5;这里只将i初始化为0，至于5的值将在初始化时赋值)，这里不包含用final修饰的static，因为final在编译的时候就会分配了，注意这里不会为实例变量分配初始化，类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。
* 解析：主要将常量池中的符号引用替换为直接引用的过程。符号引用就是一组符号来描述目标，可以是任何字面量，而直接引用就是直接指向目标的指针、相对偏移量或一个间接定位到目标的句柄。有类或接口的解析，字段解析，类方法解析，接口方法解析(这里涉及到字节码变量的引用，如需更详细了解，可参考《深入Java虚拟机》)。
* 初始化：类加载最后阶段，若该类具有超类，则对其进行初始化，执行静态初始化器和静态初始化成员变量(如前面只初始化了默认值的static变量将会在这个阶段赋值，成员变量也将被初始化)。

这便是类加载的5个过程，而类加载器的任务是根据一个类的全限定名来读取此类的二进制字节流到JVM中，然后转换为一个与目标类对应的java.lang.Class对象实例.

JVM 中内置了三个重要的 ClassLoader，分别是 BootstrapClassLoader、ExtensionClassLoader 和 AppClassLoader。

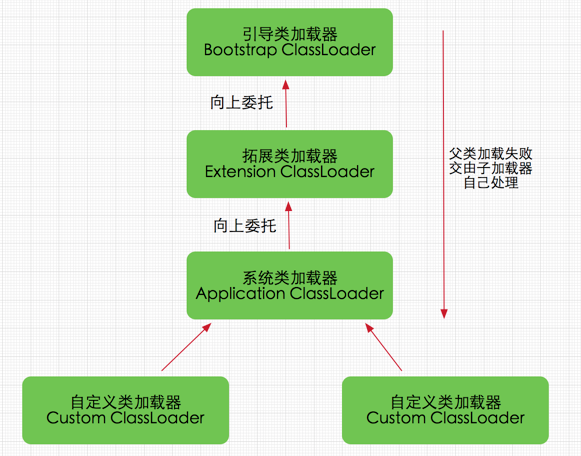
BootstrapClassLoader 负责加载 JVM 运行时核心类，这些类位于 JAVA\_HOME/lib/rt.jar 文件中，我们常用内置库 java.xxx.\* 都在里面，比如 java.util.\*、java.io.\*、java.nio.\*、java.lang.\* 等等。这个 ClassLoader 比较特殊，它是由 C 代码实现的，我们将它称之为「根加载器」

ExtensionClassLoader 负责加载 JVM 扩展类，它们的 jar 包位于 JAVA\_HOME/lib/ext/\*.jar 中，有很多 jar 包。

AppClassLoader 才是直接面向我们用户的加载器，它会加载 Classpath 环境变量里定义的路径中的 jar 包和目录。我们自己编写的代码以及使用的第三方 jar 包通常都是由它来加载的。

## 双亲委派

前面我们提到 AppClassLoader 只负责加载 Classpath 下面的类库，如果遇到没有加载的系统类库怎么办，AppClassLoader 必须将系统类库的加载工作交给 BootstrapClassLoader 和 ExtensionClassLoader 来做，这就是我们常说的「双亲委派」。



如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子加载器才会尝试自己去加载，这就是双亲委派模式，即每个儿子都很懒，每次有活就丢给父亲去干

当 parent 字段是 null 时就表示它的父加载器是「根加载器」。如果某个 Class 对象的 classLoader 属性值是 null，那么就表示这个类也是「根加载器」加载的。

Java类随着它的类加载器一起具备了一种带有优先级的层次关系，通过这种层级关可以**避免类的重复加载**，当父亲已经加载了该类时，就没有必要子ClassLoader再加载一次。其次是考虑到**安全因素**，java核心api中定义类型不会被随意替换，假设通过网络传递一个名为java.lang.Integer的类，通过双亲委托模式传递到启动类加载器，而启动类加载器在核心Java API发现这个名字的类，发现该类已被加载，并不会重新加载网络传递的过来的java.lang.Integer，而直接返回已加载过的Integer.class，这样便可以防止核心API库被随意篡改。

## 自定义加载器

ClassLoader 里面有三个重要的方法 loadClass()、findClass() 和 defineClass()。

loadClass() 方法是加载目标类的入口，loadClass()方法是ClassLoader类自己实现的，该方法中的逻辑就是双亲委派模式.它首先会查找当前 ClassLoader 以及它的双亲里面是否已经加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让自定义加载器自己来加载目标类。

ClassLoader类中并没有实现findClass()方法的具体代码逻辑，取而代之的是抛出ClassNotFoundException异常。方法是需要子类来覆盖的，不同的加载器将使用不同的逻辑来获取目标类的字节码。拿到这个字节码之后再调用 defineClass() 方法将字节码转换成 Class 对象。

defineClass()方法是用来**将byte字节流解析成JVM能够识别的Class对象**(ClassLoader中已实现该方法逻辑).

**自定义类加载器一般只会重写findClass方法。**

## Thread.contextClassLoader

主要功能是为了传递classloader,让父classloader的代码中能够获取子类的classloader的类。

我们不能使用当前类加载器的子加载器加载的类。这个限制就是双亲委派机制导致的，因为类加载请求的委派是单向的。

Thread.currentThread().getContextClassLoader().loadClass(name);

contextClassLoader **是从父线程那里继承过来的**，所谓父线程就是创建了当前线程的线程。程序启动时的 main 线程的 contextClassLoader 就是 AppClassLoader。这意味着如果没有人工去设置，那么所有的线程的 contextClassLoader 都是 AppClassLoader。

## 热部署

自定义classloader实现热部署

**符号引用和直接引用**

符号引用（Symbolic References）：

　　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，只要使用时能够无歧义的定位到目标即可。在编译时，java类并不知道所引用的类的实际地址，因此只能使用符号引用来代替。比如org.simple.People类引用了org.simple.Language类，在编译时People类并不知道Language类的实际内存地址，因此只能使用符号org.simple.Language来表示Language类的地址。符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用：

 直接引用可以是

（1）直接指向目标的指针（比如，指向“类型”【Class对象】、类变量、类方法的直接引用可能是指向方法区的指针）

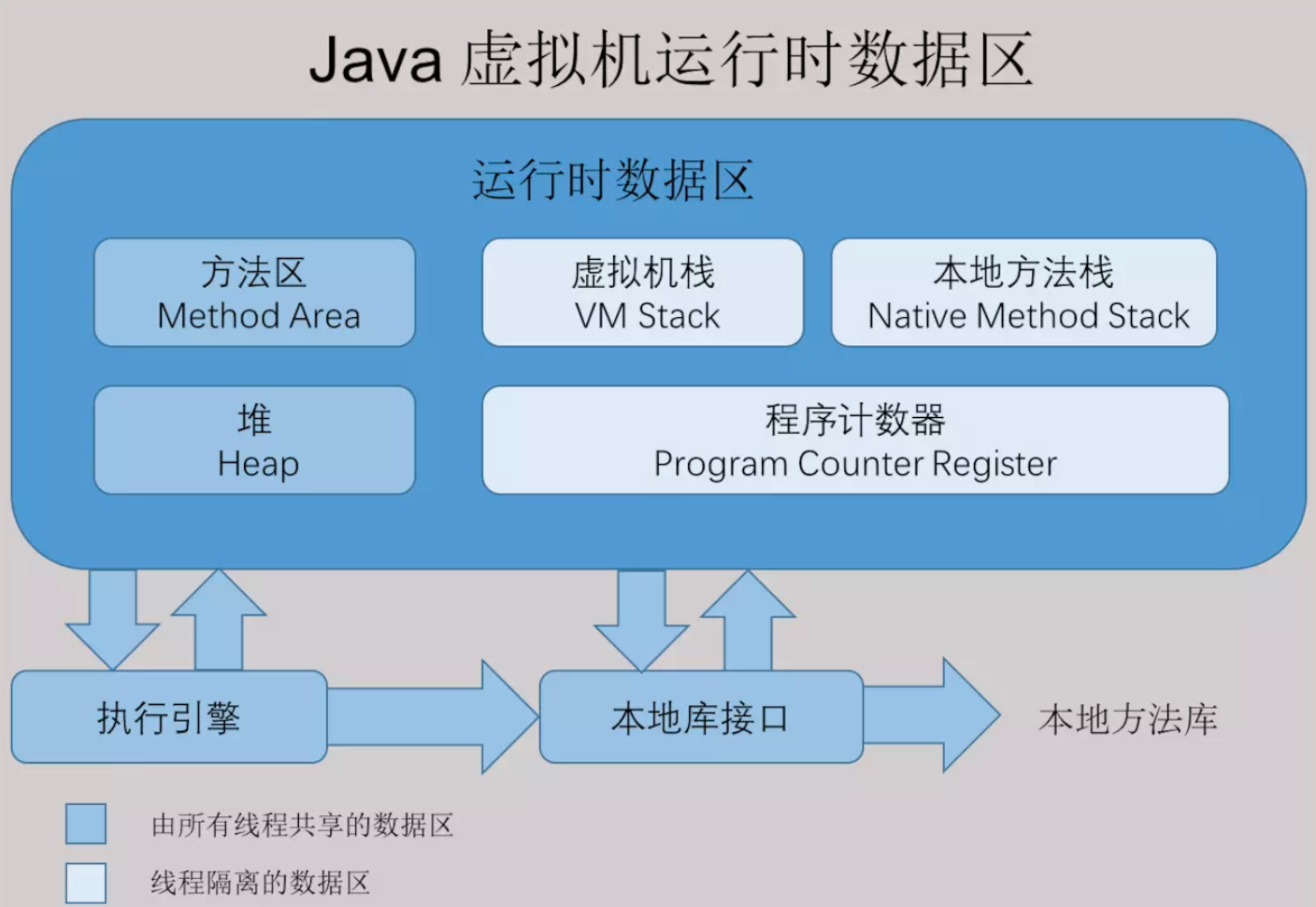
（2）相对偏移量（比如，指向实例变量、实例方法的直接引用都是偏移量）

（3）一个能间接定位到目标的句柄

直接引用是和虚拟机的布局相关的，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般不会相同。如果有了直接引用，那引用的目标必定已经被加载入内存中了。

# Java虚拟机（JVM）

**运行时数据区域**



程序计数器

线程私有。通过改变这个计数器的值来选取下一条需要执行指令.分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖计数器完成

如果线程正在执行一个 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 Native 方法，这个计数器的值则为 (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。

Java 虚拟机栈

线程私有，生命周期和线程一致。描述的是 Java 方法执行的内存模型：每个方法在执行时都会创建一个栈帧(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。每一个方法从调用直至执行结束，就对应着一个栈帧从虚拟机栈中入栈到出栈的过程。

局部变量表：存放了编译期可知的各种基本类型(boolean、byte、char、short、int、float、long、double)、对象引用(reference 类型)和 returnAddress 类型(指向了一条字节码指令的地址)

StackOverflowError：线程请求的栈深度大于虚拟机所允许的深度。

OutOfMemoryError：如果虚拟机栈可以动态扩展，而扩展时无法申请到足够的内存。

本地方法栈

区别于 Java 虚拟机栈的是，Java 虚拟机栈为虚拟机执行 Java 方法(也就是字节码)服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。也会有 StackOverflowError 和 OutOfMemoryError 异常。

Java 堆

对于绝大多数应用来说，这块区域是 JVM 所管理的内存中最大的一块。线程共享，主要是存放对象实例和数组。可以位于物理上不连续的空间，但是逻辑上要连续。

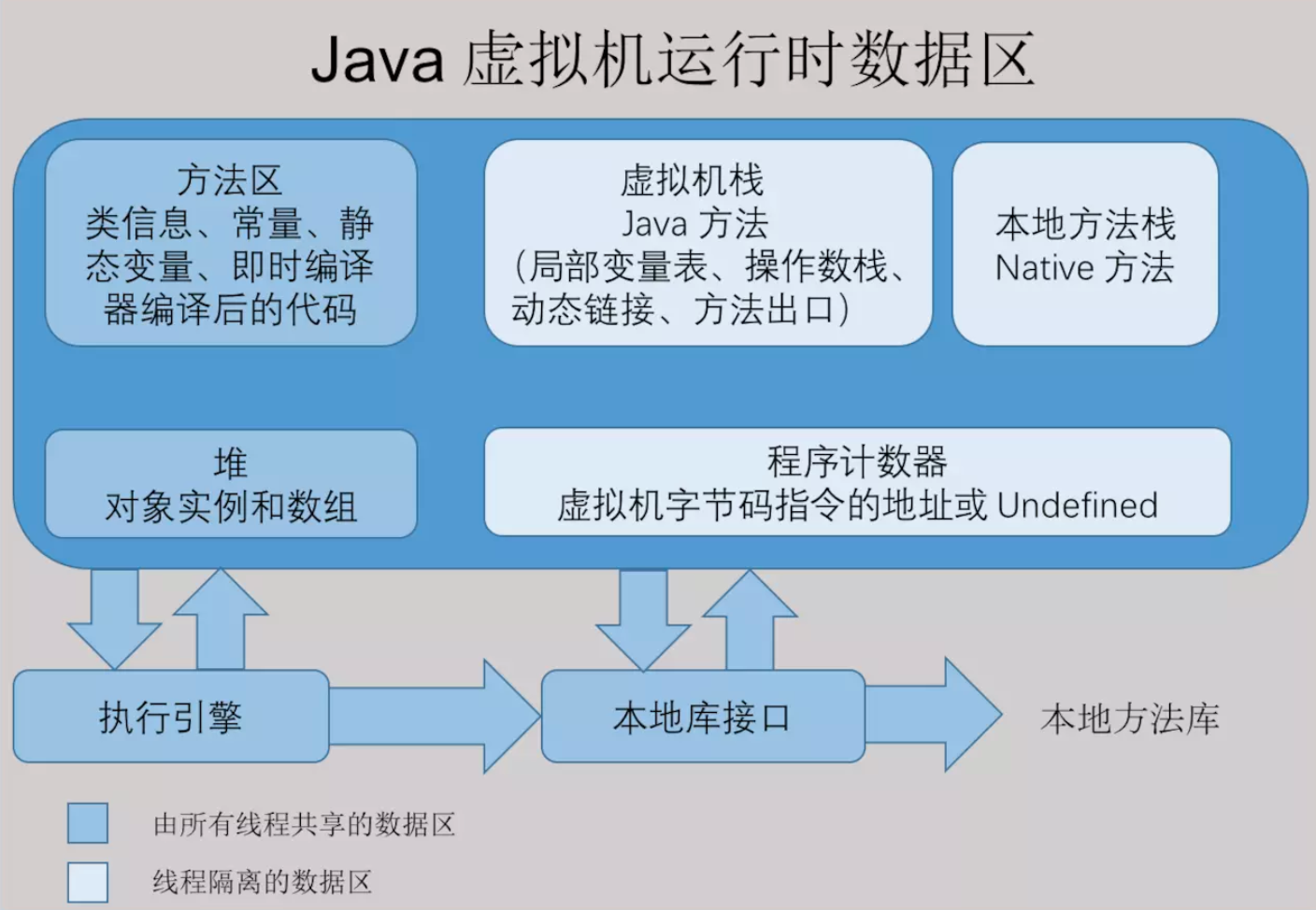
OutOfMemoryError：如果堆中没有内存完成实例分配，并且堆也无法再扩展时，抛出该异常。

方法区

属于共享内存区域，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。

运行时常量池

属于方法区一部分，用于存放编译期生成的各种字面量和符号引用。编译器和运行期(String 的 intern() )都可以将常量放入池中。内存有限，无法申请时抛出 OutOfMemoryError。



直接内存(堆外内存)

非虚拟机运行时数据区的部分，使用 Native 函数库直接分配堆外内存，然后通过一个存储在 Java 堆中的 DirectByteBuffer 对象作为这块内存的引用进行操作。可以避免在 Java 堆和 Native 堆中来回的数据耗时操作。

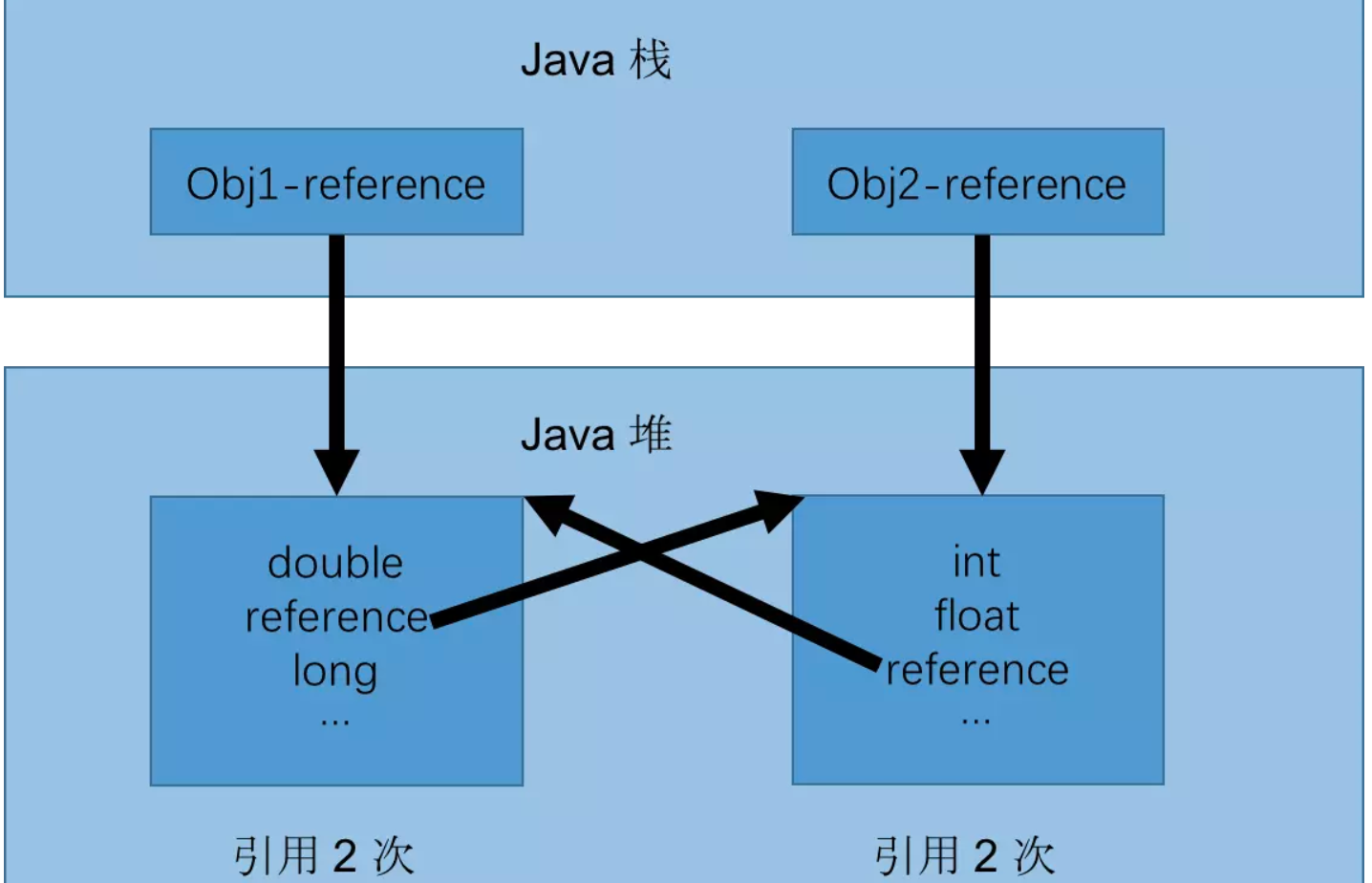
OutOfMemoryError：会受到本机内存限制，如果内存区域总和大于物理内存限制从而导致动态扩展时出现该异常。

## 垃圾回收器与内存分配策略

对象是否可以回收

**引用计数法**

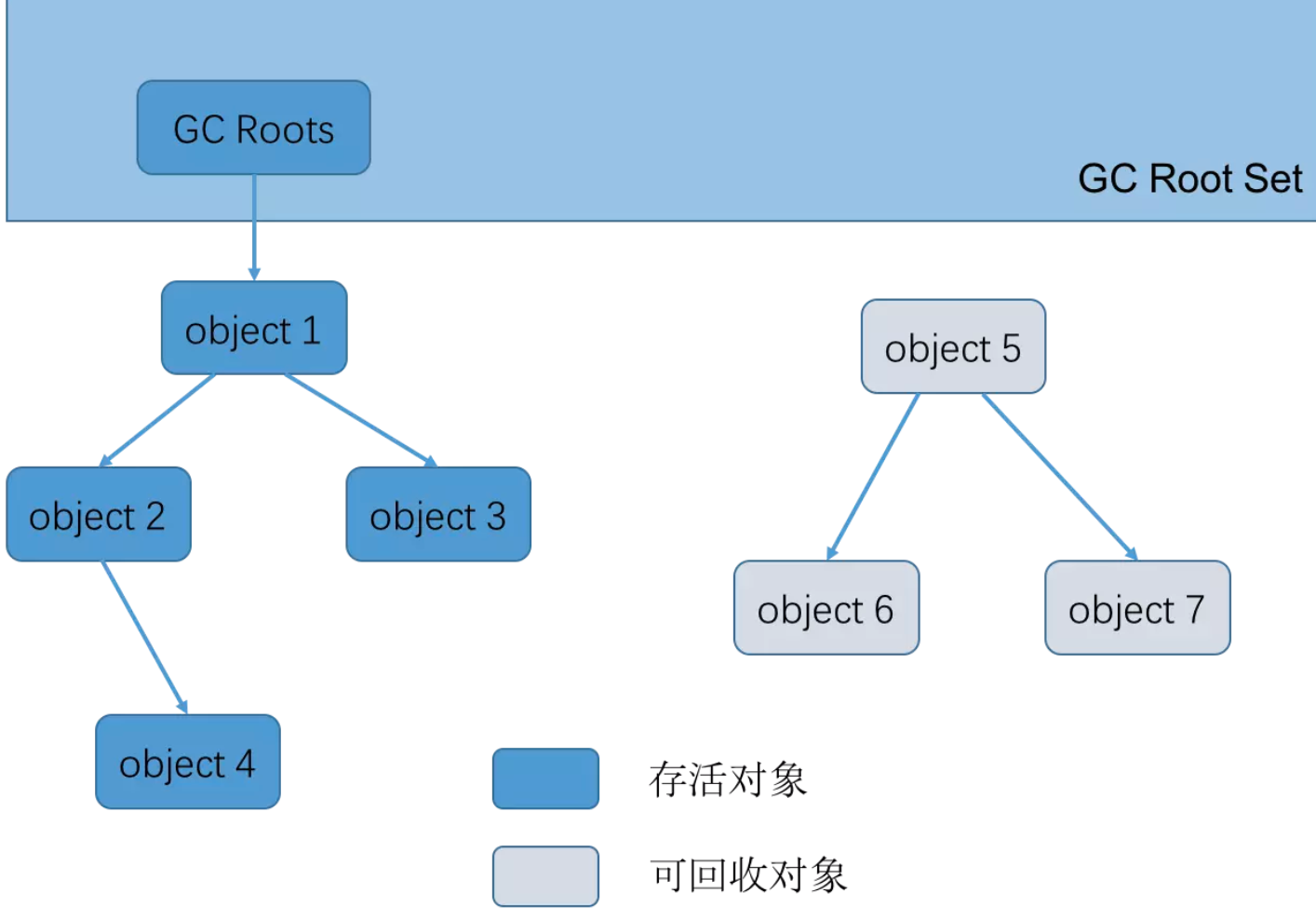
给对象添加一个引用计数器。但是难以解决循环引用问题。



直接把 Obj1-reference 和 Obj2-reference 置 null。则在 Java 堆当中的两块内存依然保持着互相引用无法回收。

可达性分析法

通过一系列的 ‘GC Roots’ 的对象作为起始点，从这些节点出发所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连的时候说明对象不可用。



可作为 GC Roots 的对象：

* 虚拟机栈(栈帧中的本地变量表)中引用的对象
* 方法区中类静态属性引用的对象
* 方法区中常量引用的对象
* 本地方法栈中 JNI(即一般说的 Native 方法) 引用的对象

**Java引用**

强引用

类似于 Object obj = new Object(); 创建的，只要强引用在就不回收。

软引用

SoftReference 类实现软引用。在系统要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行二次回收。

弱引用

WeakReference 类实现弱引用。对象只能生存到下一次垃圾收集之前。在垃圾收集器工作时，无论内存是否足够都会回收掉只被弱引用关联的对象。

虚引用

PhantomReference 类实现虚引用。无法通过虚引用获取一个对象的实例，为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。

**回收过程**

一个对象的真正死亡至少要经历两次标记过程：如果对象在进行中可达性分析后发现没有与 GC Roots 相连接的引用链，那他将会被第一次标记并且进行一次筛选，筛选条件是此对象是否有必要执行 finalize() 方法。当对象没有覆盖 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”。

如果这个对象被判定为有必要执行 finalize() 方法，那么这个对象竟会放置在一个叫做 F-Queue 的队列中，并在稍后由一个由虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。这里所谓的“执行”是指虚拟机会出发这个方法，并不承诺或等待他运行结束。finalize() 方法是对象逃脱死亡命运的最后一次机会，稍后 GC 将对 F-Queue 中的对象进行第二次小规模的标记，如果对象要在 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。finalize() 方法只会被系统自动调用一次。

### 垃圾回收算法

**标记 —— 清除算法**

直接标记清除就可。

两个不足：

* 效率不高
* 空间会产生大量碎片

**复制算法**

可以分一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 空间和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另一块 Survivor 上，最后清理 Eden 和 Survivor 空间。大小比例一般是 8 : 1 : 1，每次浪费 10% 的 Survivor 空间。但是这里有一个问题就是如果存活的大于 10% 怎么办？这里采用一种分配担保策略：多出来的对象直接进入老年代。

**标记-整理算法**

不同于针对新生代的复制算法，针对老年代的特点，创建该算法。主要是把存活对象移到内存的一端。

**分代回收**

根据存活对象划分几块内存区，一般是分为新生代和老年代。**然后根据各个年代的特点制定相应的回收算法。**

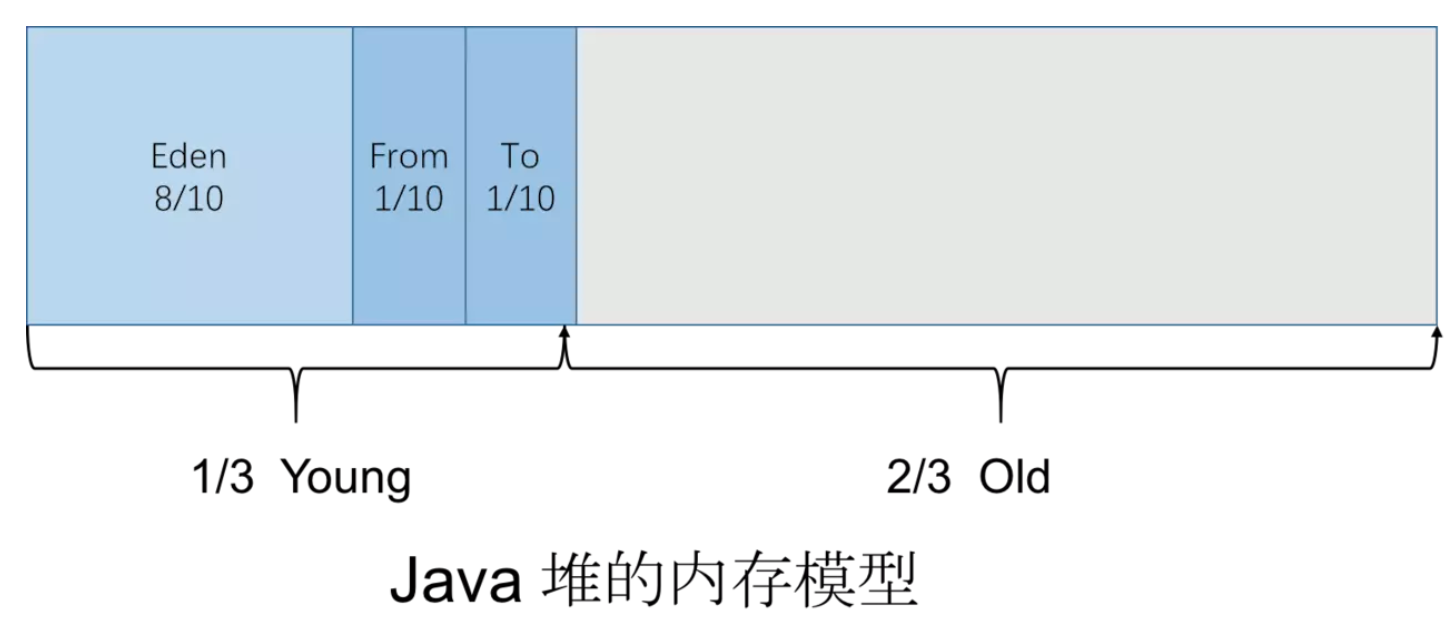
新生代

每次垃圾回收都有大量对象死去，只有少量存活，选用复制算法比较合理。

老年代

老年代中对象存活率较高、没有额外的空间分配对它进行担保。所以必须使用 标记 —— 清除 或者 标记 —— 整理 算法回收。

一般来说 Java 堆的内存模型如下图所示：



新生代 GC (Minor GC)

发生在新生代的垃圾回收动作，频繁，速度快。

老年代 GC (Major GC / Full GC)

发生在老年代的垃圾回收动作，出现了 Major GC 经常会伴随至少一次 Minor GC(非绝对)。Major GC 的速度一般会比 Minor GC 慢十倍以上。

大对象直接进入老年代

长期存活的对象将进入老年代

动态对象年龄判定

## G1垃圾回收器

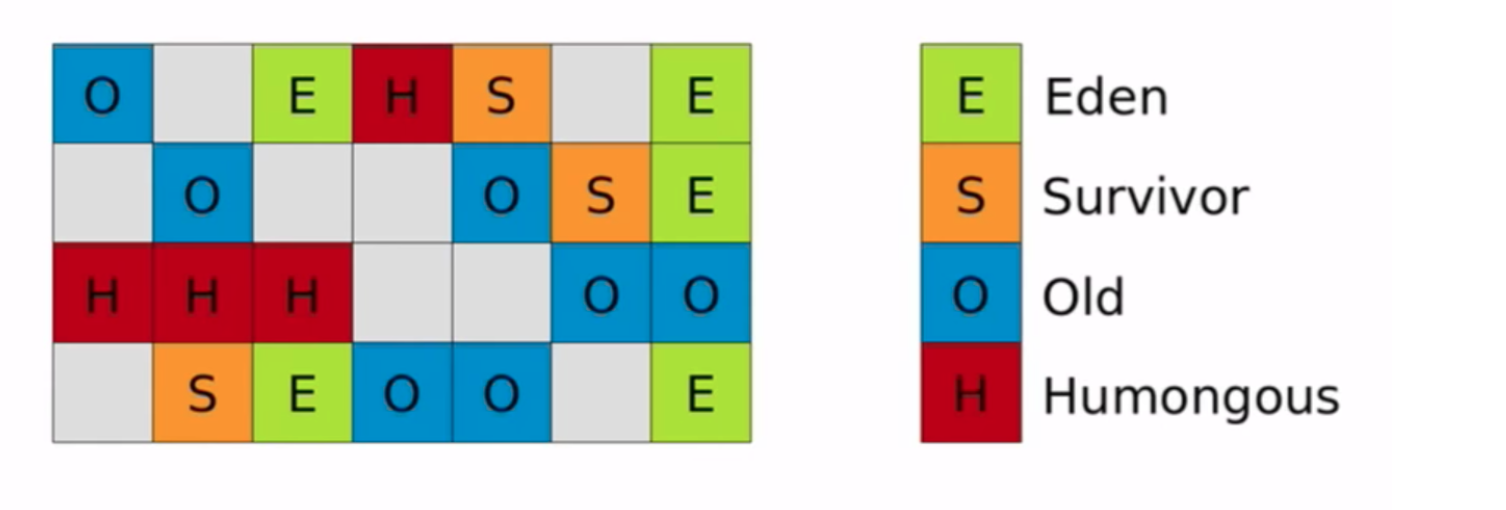
**并行性**：G1在回收期间，可以由多个GC线程同时工作，有效利用多核计算能力。

**并发性**：G1拥有与应用程序交替执行的能力，因此一般来说，不会在整个回收期间完全阻塞应用程序。

**分代GC**：与之前回收器不同，其他回收器，它们要么工作在年轻代要么工作在老年代。G1可以同时兼顾年轻代与老年代。

**空间整理**：G1在回收过程中，会进行适当的对象移动， G1在每次回收时都会有效的复制对象，减少空间碎片。

**可预见性**：由于分区的原因，G1可以只选取部分区域进行内存回收，这样缩小了回收范围，因此对于全局停顿也能得到更好的控制。



G1把堆内存分为年轻代和老年代。年轻代分为Eden和Survivor两个区，老年代分为Old和Humongous两个区。代和区都是逻辑概念。

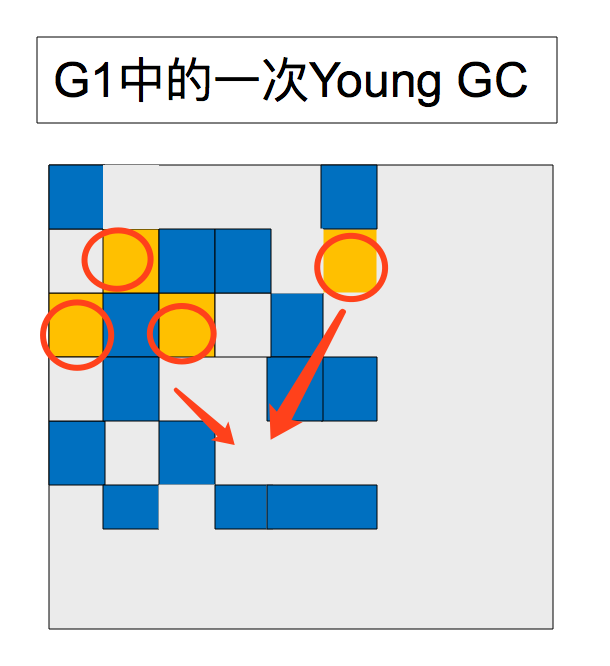
G1把堆内存分为大小相等的内存分段，默认情况下会把内存分为2048个内存分段，可以用-XX:G1HeapRegionSize调整内存分段的个数。比如32G堆内存，2048个内存分段每段的大小为16M。这相当于把内存化整为零。内存分段是物理概念，代表实际的物理内存空间。每个内存分段都可以被标记为Eden区，Survivor区，Old区，或者Humongous区。这样属于不同代，不同区的内存分段就可以不必是连续内存空间了。

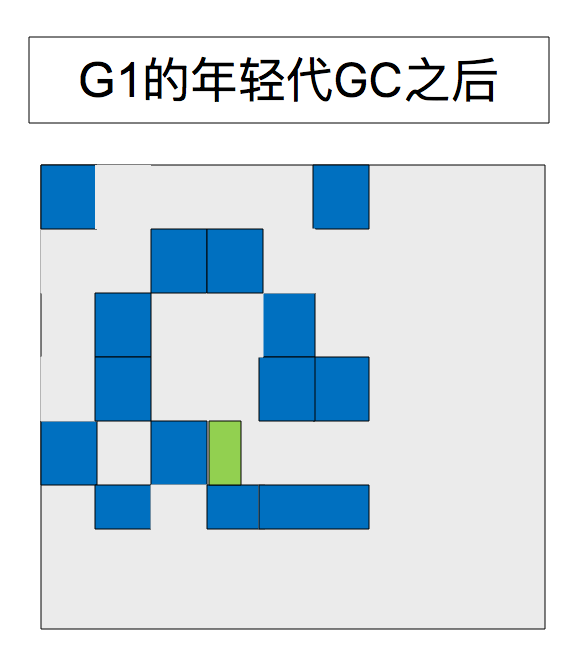
新分配的对象会被分配到Eden区的内存分段上，每一次年轻代的回收过程都会把Eden区存活的对象复制到Survivor区的内存分段上，把Survivor区继续存活的对象年龄加1，如果Survivor区的存活对象年龄达到某个阈值（比如15，可以设置），Survivor区的对象会被复制到Old区。复制过程可以达到内存整理的效果，减少碎片。

Humongous区用于保存大对象，如果一个对象占用的空间超过内存分段的一半（比如上面的8M），则此对象将会被分配在Humongous区。如果对象的大小超过一个甚至几个分段的大小，则对象会分配在物理连续的多个Humongous分段上。Humongous对象因为占用内存较大并且连续会被优先回收。

**G1中的一次Young GC**

存活的对象被转移到一个/或多个survivor 块上去。 如果存活时间达到阀值,这部分对象就会被晋升到老年代。





刚刚被晋升的对象用深蓝色表示。Survivor 用绿色表示。

总结起来,G1的年轻代收集归纳如下:

* 堆就是一整块内存空间,被分为多个heap区(regions)。
* 年轻代内存由一组不连续的heap块也就是region组成. 这使得在需要时很容易进行容量调整。
* 年轻代的垃圾收集,或者叫 young GCs, 会发生stop the world。 在回收时所有的应用程序线程都会被暂停。
* 年轻代 GC 通过多线程并行进行。
* 存活的对象被拷贝到新的 survivor 块或者老年代。

**G1的老年代垃圾收集**



老生代的G1垃圾回收有以下几个关键点：

**1、并发标记阶段（Concurrent Marking Phase）**

* 活跃度信息在程序运行的时候就被并行的计算了出来。
* 活跃度(liveness)信息标记出哪些区域块最适合回收，在转移暂停期间最适合回收掉。
* 没有sweep阶段。但CMS是有这个阶段的。

**2、重新标记阶段（Remark Phase）**

* 使用了Snapshot-at-the-Beginning (SATB)算法，这个要比CMS的算法快很多。
* 完全空的区域块会被直接回收掉。

**3、复制/清除阶段（Copying/Cleanup Phase）**

* 年轻代和老年代会被同时回收。
* 老年代的区域块会不会被选择，取决于它的活跃度。