###### 关于配置环境变量

* 1. Jdk和jre

1. jdk指的是用来开发java程序的，jre是程序运行环境，jdk中包含jre。
2. javac是编译，将我们的.java源码编译为.class字节码文件，其实就是javac.exe
3. java是运行命令，运行java.exe会让虚拟机跑起来程序
   1. path环境变量
4. 为什么要配置环境变量?

当我们从任意目录进入编译java源码的时候，例如:F:\data\javac helloworld.java时

**首先从当前目录下找javac.exe，**如果没有直接报错。

如果配置了path环境变量，如果在当前目录下找不到，就在配置的path目录下的

斌文件夹下找javac.exe

其次，我们的tomcat需要关联jdk或者jre，它会去找JAVA\_HOME这个变量

###### 面向对象

* 1. 对象的初始化过程

1. 编写一个类
2. 执行java.exe启动虚拟机.class字节码文件被加载进内存的方法区。

方法区包含:class内容区域 、静态区、方法区(构造方法)、常量池。

1. 类进入到方法区以后，先加载自己的静态成员到静态区(数据共享)
2. 运行main方法，JVM到静态区将main方法复制一份压栈执行
3. 在堆内存中创建对象，对象的一些信息
4. 对象创建完成，支持方法

###### 集合

* 1. 为什么要使用集合？

1. 主要是为了解决扩容问题，arraylist的默认长度是10，当存储的数据长度超过10就

可以进行自动扩容

3.2几种常用集合的比较

1、ArrayList：底层是数组，查询快，增删慢。

2、LinkedList:底层是链表,查询慢，增删快。

以上两个都是继承list接口

3、HashSet底层是hashmap，底层数据结构是数组+链表结构。

4、TreeSet，底层是二叉树

以上两个集合都是继承set接口

1. map集合的底层数据结构是hash表，是一个数组+链表的结构

3.3关于Iterator并发修改异常的底层机制

1、AbstractList中有一个变量叫做modCount，它是用来记录对集合进行操作的变量，每次增删改查都会对变量进行加1操作。

2、关于迭代器如何使用此变量

在遍历集合之前，先将modCount进行复制，在取值之前，即调用hasNext()方法，如果modCount发生改变，直接会抛出异常。

3.4 ArrayList去重原理

1、先创建旧的容器

2、再创建新的容器，遍历旧的集合容器，如果新的集合中有这个元素，就不需要增加这

个元素。

3、如何判断集合中是否含有此元素？

调用集合的contains方法来判断集合中是否含有这个元素，而最终contains方法还是调用对象的equals方法来进行判断。

4总结ArrayList去重的原理，类重写equals方法

3.5 HashMap的存储数据解析

1. hashMap的底层数据结构为hash表，是一种数组+链表+红黑树的结构类型
2. hashMap存储数据的流程？
3. 首先获取改数据的hash值
4. 在hash表中判断是否冲突，如果不冲突，直接插入元素，也就是在数组中插入元素
5. 如果冲突即hash值是相同的，则在相同的hash值插入在数组相同的位置，这个时候将插入的元素转换成链表或者红黑树的结构。
6. 如果是链表结构，进行判断，如果插入后的元素大于8，转为红黑树再进行插入
7. 如果不大于8，则插入链表的末端。

3.6-关于Arrays.asList()

1、Arrays.asList()的底层是数组，所以不能对其调整尺寸的操作

2、Arrays.asList方法接收一个数组或者一个用逗号分隔的元素列表

###### 泛型

4.1-泛型类

1. 在类上定义泛型，public class object<T>{}
2. 在创建对象后，用户可以指定泛型的类型，比如整型、字符型等
3. 类上申明的泛型只对非静态成员有效

4.2-一个元组类库

1、仅一个方法调用就能返回多个对象的时候，我们有几种方法?

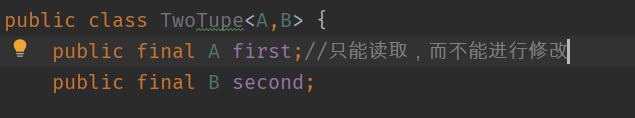
A、创建一个对象，包含方法所要返回的对象类型。

缺点:每次在需要的时候，都要进行创建对象

B、创建元祖，格式如:TwoTuple<A,B>:包含两种类型的元组

C、ThreeTuple<A,B,C> extends TwoTuple<A,b>-包含三种类型的元组。

2、关于类成员变量



当在类中申明成员变量是 public final的时候，当创建一个对象的时候，我们只能对其进行读的权限，而不能进行写的操作。如果想改变其值，我们必须重新创建一个新的对象。

4.3-泛型应用于接口上

Public interface Generator<T>{ T next();}

创建一个类实现Generator<T>接口和Iterable接口，generator接口用于产生单个对象，而实现Iterable接口，可以在for each的循环中使用迭代器。

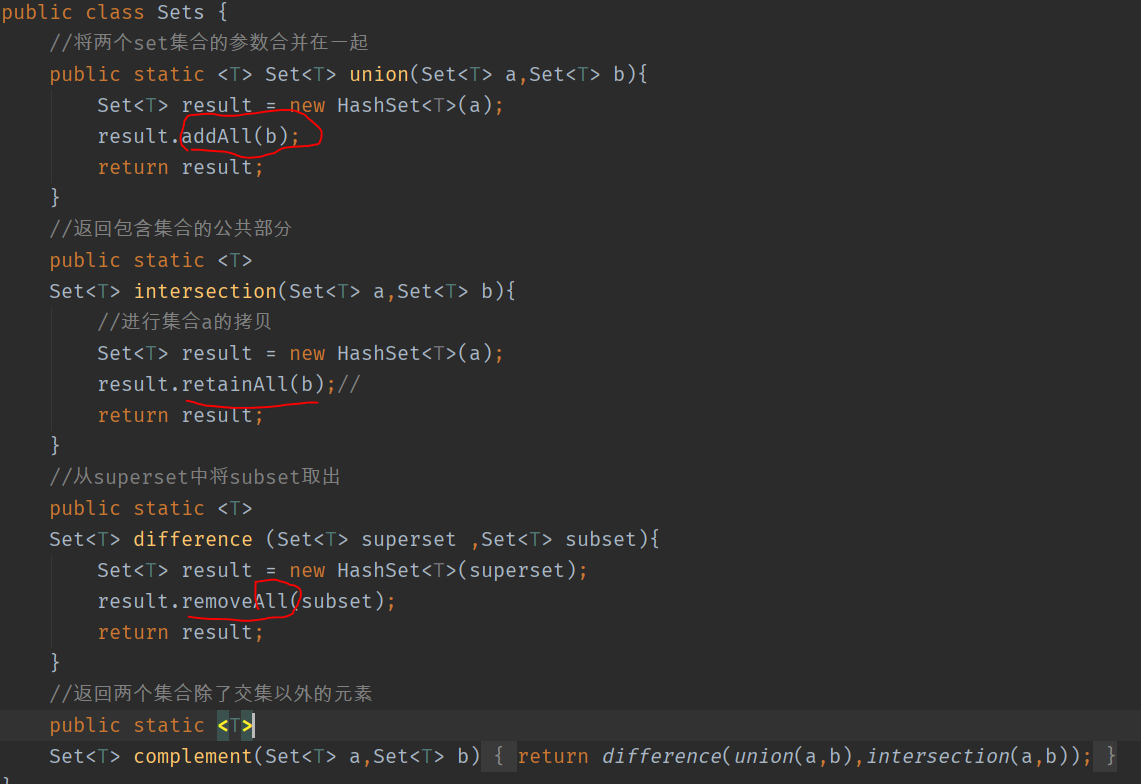
4.4-泛型应用在方法上

1、泛型应用在方法上的格式

Public <T> void f(T x);

2、原则就是如果可以使用泛型方法取代整个泛型类，那么只需要使用泛型方法

3、生成一个set集合的工具方法



4.5-利用泛型构建复杂的模型

4.6-泛型的擦除

1、由于泛型只在编译期进行类型的检查，在运行的期间都会进行泛型的擦除。任何在运行的期间，获取确切的类型的操作都将无法工作。

2、使用instanceof最终会失败，因为确切的类型会被擦除。我们可以使用动态的isInstace()方法

3、当使用newInstance()方法创建对象时，有个前提条件就是这个类必须含有空参构造。

4.7-通配符

1、？表示通配符

List<? Extends Fruit> -表示任何从Fruit继承的类型的列表,但是不能调用add方法添加任何对象，因为不知道具体的添加的类型，所以禁止添加任何类型的元素。

协变-表示这个是这个类或者这个类的子类，无法确定任何类型，所以不允许添加任何对象或者元素或者进行set方法的设置元素。

2、List<? Super T>表示的是逆变，T是基类，可以向list添加T或者T的子类

4.8-泛型数组

1、泛型数组，数组会跟踪他们的实际类型，而这个类型会在数组被创建的时候确定。如果使用泛型的方式，但这个信息只会存在编译期的时候，在运行的时候，由于擦除，数据会当作成Object[]类型。

###### 5. IO流

5.1 File类

5.11-构造方法

Public File(String pathName);//路径可以是绝对路径和相对路径

5.12-常用的方法

Public String getAbsolutePath();//获取该文件的绝对路径

Public boolen exists();//判断该文件是否存在

Public Boolean IsDirectory();//判断file是否为目录

Public boole isFile();//判断此file是否为文件

Public Boolean createFile();//仅当该文件不存在时，创建该文件

Public Boolean mkdir();//创建该文件的目录

Public Boolean mkdirs();//创建该文件的多级目录

5.2 IO的分类

按照流向分类:输入流和输出流

按照数据的类型分:字节流和字符流

硬盘到内存--------输入

内存到硬盘--------输出

5.3 IO的继承体系

一切皆为字节流

InputStream-FileInputStream

OutputStream-FileOutputStream

5.4 字节输出流（OutputStream）常用方法

5.4.1 FileOutputStream的构造方法

Public FileOutputStream(File file);//通过指定file对象，创建输出流

Public FileOutputStream(String name);//直接指定路径名称

Public FileOutputStream(File file,boolean append);//创建输出流，设置是否追加数据

Public FileOutputStream(String name,boolean append);//设置是否追加数据

“\r\n”----换行符号

5.4.2 常用方法

Public void close();//关闭流

Public void flush();//将缓冲区中的数据写入到硬盘中

Public void Write(byte[] b);//将指定的字节数组写入到此输出流

Public void Write(byte[]b,int off,int len);//从偏移量off开始，将指定的len长度写入到

5.5字节输入流（InputStream）常用方法

5.5.1 FileInputStream的构造方法

FileInputStream(File file);

FileInputStream(String name);

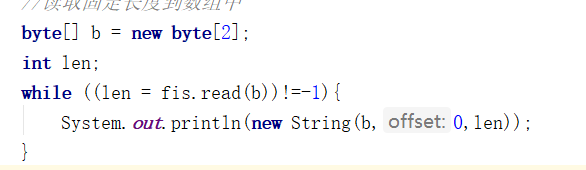
创建文件输入流进行数据读取时，文件必须存在，否则会出现异常报错

5.5.2 常用方法

Int read();//读取单个字节，当读取到末尾时返回-1，通过while循环来读取数据

Int read(byte[] b);//读取固定长度的字节数到数组中，**返回值是读取到的字节的长度**

**每次读取新的内容覆盖之前读取到的旧的内容,所以可以通过读取长度来获取**

****

**建议开发中使用数组进行读取，减少io读取速度，提高开发效率**

###### 6.多参数的构造方法建议使用bulider内部类进行实现-21

###### 7.要优先使用基本类型而不是包装类，基本类型免除了装箱的过程

###### 8.只要是类自己管理内存，程序员就要警惕内存泄漏-32

###### 9.try\_with\_resource

Try\_finally—如果try中也抛出异常，finally也抛出异常，那么第二个异常会覆盖第一个异常。

Try\_with\_resource如何使用？

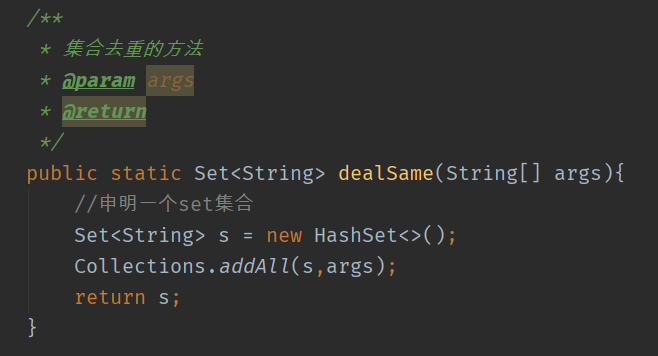
要想使用这个方法，被释放的资源必须实现AutoCloseable接口。。。因为只有实现这个接口才会调用close方法释放资源

写法为try(){}catch(){}，要释放的资源在try()中创建，catch捕获处理异常。

这个方法的好处就是业务处理和关闭连接都出现异常，会打印业务处理的异常。而关闭连接的异常通过getSuppressed()方法进行释放

###### 10.覆盖了equals方法，那么必须覆盖hashCode方法，尽量每个类覆盖toString方法

###### 集合进行去重



集合去重实现依赖于String 类实现Comparable接口

###### comparable接口

总而言之，对排序比较敏感的时候，都英爱让这个类实现comparable接口，以便其实例可以被分类、搜索以及应用基于比较的集合中。每当在compareTo方法的实现中实现比较值中的时候，都要避免使用<和>操作符，应该在装箱的基本类型中使用静态方法compare方法

###### 类和成员可访问的最小化

1. 如果类或者接口能被做成包级私有的，那么就应该做成包级私有的。把这个类做成包级私有的，实际上这个类成为了这个包实现的一部分，而不是这个导出的API的一部分，在以后发布、修改、替换而无须担心客户端的版本的兼容性。

###### 复合优先于继承

1. 在包内部使用继承是非常安全的，因为子类和超类都是同一个程序员在维护。
2. 对于普通的类进行跨越包边界的继承是非常危险的。

###### 访问权限控制

14.1-权限修饰符

1、当方法或者是成员变量省略了权限修饰符的话，表明这个只对包类有权限，对包外的类是无权限进行访问的。

2、protected权限修饰符针对子类继承父类，当希望父类某些特定的成员赋予子类的访问权限的时候，而不是所有的类的时候，就需要使用protected。

3、一个类的访问权限只有两种:private和public，如果不希望其他任何类对该类有权限，那么可以让这个类的构造器指定为private。

###### 15.复用类

15.1-空白final

1、所谓空白final是指在声明为final，但又未给定初始值，无论什么情况下，编译器都必须确保空白的final域在使用前必须被初始化，一旦被初始化，final域就无法被修改。

2、覆盖只有在某方法是基类接口的一部分时才会出现。如果某方法是private那么这个方法就不是接口的一部分。它只是隐藏于类中的程序代码，如果你在子类中”仅具有相同名称”的情况，此时你并没有覆盖方法，而是生成一个新的方法而已。可以使用@Override的注解可以提示是否是覆盖方法。

3、类的字段在定义的时候，赋一个初始化的值。如果构造函数，是没有对字段重新赋值的话，那么创建的对象会使用初始化的值。否则的话，创建的对象会使用构造函数赋予的值。

###### 16.内部类

16.1-内部类的使用

1、如果你想在接口创建一部分的公共方法，而被所有的实现类所使用，那么你可以创建一个接口内部类。

2、内部类的使用场景-控制框架

3、内部类分为局部内部类和匿名内部类，局部内部类有名称，可以有相应的构造函数。而匿名内部类大多数是针对接口，所以是没有构造函数。并且匿名内部类对象只能存在一次，而局部内部类对象可以有多次出现的情况。

###### 17.反射

17.1-再谈初始化

1、仅使用类.class，获取类的字节码的class对象是无法对类进行初始化的。而Class.forName()是进行初始化动作。

2、如果一个域值是static final，并且是编译期常量，那么这个值是可以不被初始化而被读取的。如果这个域仅仅是static和final而不是编译器的常量，那么必须进行初始化才可以进行访问

3、如果一个static域不是final的，那么对它的访问必须是进行初始化的。

###### 18．容器(持有对象)

18.1-关于迭代器

1、数据和Collection对象可以使用for each主要是由于实现了Iterable方法，该接口有包含产生Iterator的方法，for each方法会使用默认实现的迭代器，例如ArrayList等一些集合

也可以自己进行适配实现迭代器的接口方法

2、for each可以用于数组或者其他任何Iterable，但是数组不是一个Iterable。

如果要实现数组也是，必须对数组进行手动的转换Arrays.asList(数组);

###### 19.泛型-混型

1、混型-就是一个类可以有多得类的特性，我们可以通过这个类来使用他们的方法。而由于java没有多重继承的概念我们必须通过其他方式实现混型

19.1-混型实现方式1-静态代理

1、通过一个类实现不同的接口

2、在这个类有相应接口的域

3、在这个类编写所需要的方法，通过域的方法去实现静态代理。

19.2-混型实现方式-2-动态代理

1、java动态代理-每个被混入的必须是某个接口的实现

2、动态代理代理类必须实现InvocationHander接口，这个接口的抽象方法invoke必须是被覆盖重写的，代理对象的所有目标对象的方法都是要经过invoke方法。

3、Proxy有个静态的方法 newProxyInstance()接收三个参数:

A、一个类加载器（因为代理的是接口，接口没有构造函数，必须传递类加载器）

B、要实现的接口数组，代理对象实现所有需要的实现的接口，所以可以称为混型

C、InvocationHander的实现类，主要是实现Invoke方法

###### 20.泛型-潜在类型

20.1-潜在类型机制概述

1、潜在类型语言只要求实现某个方法的子集，而不是特定的类或者接口，即我不关心你是什么类，只关心的你是否具有某个特定的方法，例如speak方法

20.2-JAVA实现潜在类型机制-反射

1、利用反射机制，在反射中，通过getMethod获取特定的方法，实现潜在的机制

2、反射的缺点就是-反射是在运行期间进行检查，编译期间无法判断这个对象是否具有这个方法。

20.3-JAVA实现潜在类型机制-适配器模式

1、潜在类型意味着，不关心你的类的类型，只要他们具有具有这些方法即可，那么可以通过适配器模式实现潜在类型

2、适配器定义一个接口，接口包含所需要的方法。

3、创建适配的类继承现有的类，实现适配器定义的接口，我们在方法调用时，只接收适配器定义的接口。

###### 21.容器

21.1-关于可选操作

1、Collection接口的某些方法是可选的，即在实现的子类 中不一定要实现。这样做的好处，防止接口爆炸，如果调用了没实现的接口方法，会抛出unsupportedOperationException的异常提醒。

2、Arrays.asList()方法返回一个数组，数组就是其数组的大小不可以改变的底层结构，数组的大小不可改变，但是里面的值是可以改变的。

3、Collections.unmodifiableList()方法将产生不可改变的数组，包括这数组的值和数组的大小。

4、对于将容器作为参数的接收的方法，其文档应该指定那些可选的方法必须实现。

21.2-关于set集合

1、存入set的每个元素必须是唯一的，因为set不保存重复的元素。加入的set的元素必须定义hashCode方法和equals()方法保证元素的唯一性set接口不维护元素的次序。

2、HashSet为快速查找而设计的set。存入HashSet的元素必须定义hashCode()方法

###### 22.多线程

22.1-可以使用线程池管理线程

ExecutorService exec = Executors.newCacheThreadPool();//创建线程池

ExecutorService exec = Executors.newSingleThreadPool();//创建唯一一个线程

如果是唯一的话，任务将以排队的方式进行等待任务被执行

exec.execute(实现runnable的类，即所谓的任务)

22.2-多线程可以有返回值

1、使用Runable接口实现的任务是没有返回值的，如果要有返回值，必须使用Callable接口

22.3-关于后台线程

1、当所有非后台线程结束后，程序也就终止了，同时会杀死所有的后台线程

2、任何非后台线程还在运行，程序就不会终止。

###### 23-JavaWeb的三大组件

23.1-Sevlet

1、生命周期的三个方法:init()、destory()（以上的方法的只执行一次）

Service的方法可以多次执行

1. servlet是单例的,不是线程安全的
2. 类由我们自己创建，对象是服务器创建，方法是服务器调用
3. SevletConfig对应的是配置文件的对象，可以获取上下文数据对象

23.2-关于HttpServlet

1、HttpServlet是继承了GenericServlet(该类主要是获取了sevlet的配置和上下文对象信息)

2、关于servlet的执行流程

A、tomcat请求servlet服务

B、执行周期方法

C、执行service(ServletRequset,ServletResponse)方法会将参数强转为HttpServletRequset和

HttpServletResponse,然后会调用重载的Service(HttpServletRequest,HttpServletResponse)方法。

D、最后改方法会根据请求的方式，执行相应的方法，例如:doGet()、doPost()

E、我们自己继承了HttpServlet只需要覆盖重写doGet()或者doPost()方法，如果没有进行覆盖重写，则会报一个405错误异常。

23.3-Servlet细节

1、servlet不是线程安全，效率快，但是有线程安全的隐患

2、可以放没有状态、或者是只读状态的数据

23.4-SevletContext

1、ServletConfig、Httpsession、SevletListnerEvet----都可以获取ServletContext对象

23.5-观察者模式

1、监听器使用观察者模式，观察者模式有三要素

A、事件源-小偷

必须有注册监听的方法，注入监听

在发生某些事情的时候，调用监听器的方法

B、事件-

包含事件源的类

C、监听器-警察

是个接口，我们自己去实现，需要被注册

必须在某些特定的方法执行的时候被调用

23.2-javaWeb中的监听器（8大监听器）

事件源:

1. ServletContext
2. 生命周期监听-创建到销毁-servletContestListner
3. 属性监听-属性的增加、修改、删除的监听-servletAttributeListner
4. HttpSession
5. 生命周期监听器 -HttpSessionListner
6. 属性监听器-HttpSessionAttributeListner
7. ServletRequest
8. 生命周期监听器-ServletRequestListner
9. 属性监听器-ServletRequestAttributeListner
10. 感知监听器（与httpSession有关）
11. 他是用来注册到JavaBean上，而不是添加到三大域中
12. 这两个监听器不需要在web.xml中进行注册
13. javaBean如果实现了这个监听，就可以感知这个类似是否在session域中存在了
14. HttpSessionActivationListener

Session-钝化，把对象序列化转换为字节，保存在硬盘中

Session-活化，把对象反序列化转换为对象，读进内存中

实现改监听的对象，必须实现序列化接口

###### 24-关于数据库和JAVA的时间类型Data

1. 数据库的

Date – java.sql.Date

Time-java.sql.Time

TimeStamp-java.sql.TimeStamp

1. 领域对象不可以出现java.sql包下的东西，即java.sql.Date
2. ResultSet.get()---获取的java.sql下的时间类型

PrepareStament setDate(int n,Date) 的第二个参数也是java.sql

1. 从数据库读取的时候 直接可以使用util.Date进行接收

因为sql.Date是util.Date的子类 类型的向上转换是OK的

同样Time、TimeStamp都是Date的子类

1. 从java到数据库必须是java.sql.Date的数据类型

可以通过获取Date下的毫秒值

然后再new sql.Date(new Date().getTime())//转换为毫秒值

###### 25-关于类加载器

25.1、类加载器的分类

1、引导:类库

2、扩展:扩展jar包

3、系统:应用下的class，classpath目录下的类

25.2-类的加载遵循委托机制

1、第一步:通过findLoadedClass()方法，查询类是否被加载，防止重复加载出现

2、如果返回是null，调用上级的类加载器，

3、如果上级没有返回，那么只有自己去加载这个类，如果找不到类，就会返回classNotFound的错误异常。

###### 26.装饰者模式

26.1-对象的增强模式

1、继承

被增加的对象是固定的

增强的内容是固定的

2、装饰者模式

被增强的对象是可变的

被增强的内容是固定的

3、动态代理

被增强的对象是可变的

被增强的内容是可变的

26.2-装饰者模式

1、装饰者模式的使用场景

装饰:不知道被增加的对象的具体类型的时候，可以实现。即这个对象的类有共同的接口或者父类。

1. 装饰者的特点

是你有你，一切拜托你

Is a

Has a

Use a

1. 举例说明

Connection con = ………

Public MyCollection implements Connection{//是你

Private Connection con;//有你，被增加的对象

Public MyCollection (Connection con){//通过构造器传入需要被增强的对象

This.con = con;

}

//一切拜托你，

Public Stament createStament(){

Return con.createStament();

}

//增强点

Public void close(){

//

把当前的连接归还到池中

}

}

###### 27.动态代理模式

**1、动态代理作用:**

**在运行的时候，动态创建一组指定的接口的对象**

**2、动态代理的三大参数**

**Object proxy = Proxy.newProxyInstance(loader,new Class[] interface,new InvocationHander)**

**Loader:类的加载器**

**Interface:实现的接口数据，interface a.class**

**InvaocationHander-接口所有的动态代理对象的方法会执行Invoke方法**

**3、关于InvoationHander**

###### 28.多线程

28.1-关于interrupted方法

28.1.0-API的概述

Yield-方法是使线程处于就绪状态

Join()方法是线程同步等待的方法

28.1.1-打断阻塞线程

1、线程在阻塞状态，例如:执行sleep、wait或者join方法，可以被打断，抛出异常

2、当执行完打断的方法，会清空打断标记()

28.1.1-打断正在运行的线程

1、不会清除打断标记

2、打断标记还是true，用这个可以退出当前的任务

28.1.3-两阶段的终止模式

1、在正常运行的时候，打断标记不会被清空

2、在sleep的时候打断，会被清空打断标记，需要在异常处理代码中再次设置打断标记

3、静态方法interrupted不会清除标记

28.1.4-关于守护线程

1、何为守护线程？

守护线程就是在没有线程在运行的时候会强制退出

2、如何设置线程为守护线程?  
 t1.setDaemon(true);

默认是非守护线程

28.2-线程的状态

1、线程的创建

2、可运行状态(就绪状态)，可以被操作系统调用，但是没有获得cpu时间片

3、运行状态，获得cpu的时间片

4、阻塞状态，例如:读写文件，放弃cpu的使用权，调度器不会考虑阻塞状态线程

等操作系统唤醒，可能不会立即执行，等待操作系统分配时间片

1. 终止状态，不会再转换为其他状态

28.3-synchronized关键字

1、关于synchronized的关键字，他保证了执行代码的原子性，即便遇到多线程的上下文切换也会保证代码的原子性执行。

2、关于synchronized加在普通得非静态方法上，关联得锁是this对象

3、关于synchronized加静态方法上，关联得是该类得.class对象。

4、调用sleep方法并不会释放锁对象

28.4-关于局部变量得线程安全问题

1、方法体得局部对象不会有线程安全，每个线程在占内存创建自己得存储空间，互不干扰

2、当方法引用得是对象得引用，有可能存在线程安全，因为对象实在堆中创建唯一得对象引用

3、尽量在方法中使用private修饰符，一定程度上加强线程安全

28.5-关于Monitor

28.5.1-Monitor对象头

1、普通对象的组成部分

A、标记部分(32位)

B、class指针(32位)-指针指向对象

2、数组对象

比普通多一个数组长度(32位)，指定数组的长度

28.5.2-Monitor的底层原理

Thread3

Thread2

Thread1

Owner

EntryList

WaitSet

Monitor

1. 每一个对象锁会关联一个Monitor对象、
2. 一开始的时候Monitor的owner为null
3. 当Thread-3执行到Synchornized代码块时，就会将monitor的所有者设置为Thread-3
4. 在Thread-3上锁的过程中，如果Thread2、Thread1也来执行上锁的代码快，那么就会进入阻塞的队列EntryList（底层是双向链表）

28.5.3-轻量级锁

1、使用场景:如果一个对象虽然是多线程访问，但是访问的时间是错开的，即不存在竞争的情况，那么就可以使用轻量级锁

2、轻量级锁对使用者是透明的，即语法还是synchronized

3、如果是在同一个对象的同一个线程下加锁，会出现锁重入的情况发生，这时候会通过的线程的锁记录的个数来确定加索的个数。

4、解锁的时候，会先消除重复的锁记录，然后将对象的hashCode等对象特性返回给加索的对象

28.6-锁膨胀

1、如果在尝试加轻量锁的过程中，CAS操作无法成功，这时有一种情况就是其他线程已经对其加锁，存在竞争，这事需要进行锁膨胀，将轻量级锁变为重量级锁

28.7-自旋优化

1、重量级锁竞争的时候，线程不会立马进入阻塞的状态，会在外循环等待锁被释放，如果被释放了，就可以不用阻塞，直接获取对象锁。

注:这种对单核cpu没有啥意义，因为自旋会一致占用的cpu的资源

28.8-偏向锁

28.8.1-偏向锁的概念

1、轻量级锁在每次没有竞争的时候就自己的线程，每次重入都会执行CAS的操作

2、java6以后对其进行优化，即偏向锁，只在第一次使用CAS将线程的ID记录在对象的markword头，之后如果发现这个线程ID是自己没有竞争，不用重新的CAS操作。以后只要不发生竞争，这个对象就归这个线程所有。

28.8.2-偏向锁的状态

1、如果开启了偏向锁（默认开启），那么创建对象后的markWord的后三位就是101，这时的thread、epoch、age都为0

2、偏向锁是默认延迟的，不会在程序启动时立即生效，如果想避免延迟，可以加vm参数来避免延迟加载

3、如果没有开启偏向锁，markWord后三位为001，hashcode、age都为0，第一次使用到hashCode值才会赋值

4、可以设置禁用偏向锁，设置参数，当调用对象的hashCode的方法时禁用偏向锁，因为偏向锁没有额外的空间存储hashCode。而轻量级锁有线程的栈帧存放hashCode、重量级的有monitor对象存放hashCode值

5、加锁的顺序:偏向锁、轻量锁、重量锁

28.8.3-偏向锁的撤销

1、当有其他线程竞争的时候（没有交涉，错开），会撤销偏向锁升级为轻量锁

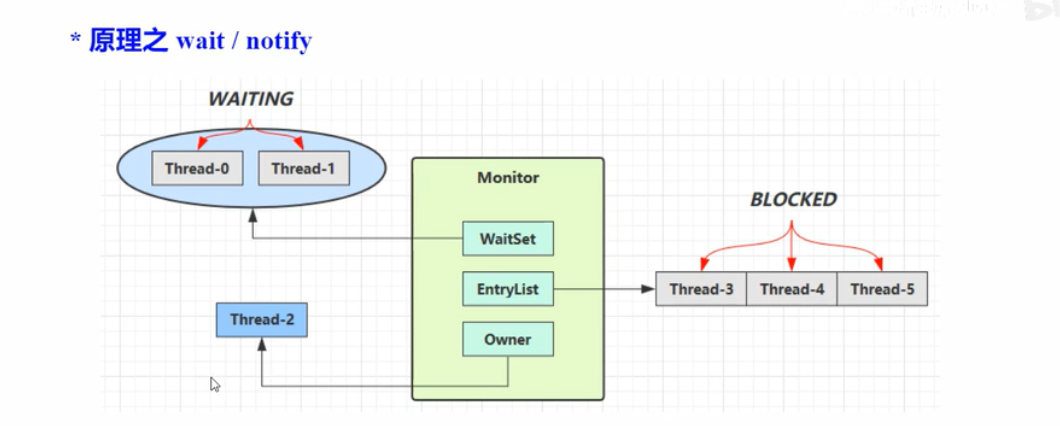
28.8.4-批量重偏向

1、如果对象被多个线程访问，但没有竞争，这时偏向的线程可以由T1线程切换为T2线程，对象的设置为T2的线程ID

2、当撤销超出20此的时候，JVM会觉得我是不是偏向错了线程，于是会在这些加索的对象重新偏向至加锁线程

28.9-wait/notify

28.9.1-wait/notify原理



1. Owner发现条件不满足，调用wait方法，进入 WaitSet，变为waiting状态
2. Blocked和Waiting状态的线程都处于阻塞状态，不占用cpu的时间
3. Blocked线程会在Owner线程被释放的时候唤醒
4. Waiting线程会在owner线程调用notify或者notifyAll时唤醒，但唤醒后不会立即获取锁，仍需要进入EntryList进行重新竞争锁。

28.9.2-wait/sleep

1、sleep(long n)和wait(long n)的区别

A、sleep是Thread的静态方法而wait是Object的一个方法

B、sleep不需要配合sychronized使用而wait必须配合sychronized进行使用

C、sleep在睡眠的时候不会释放对象锁但线程在wait的时候会释放对象锁

2、共同点:线程状态是一样的，都是timeWaiting

3、我们在定义公共锁对象的时候，可以加上修饰符final来保证对象的不可变，保证线程拥有的是同一个锁 static final Object lock = new Object();

28.9.3-wait/notify的套路写法

Sychronized(lock){

While(条件不成立){

Lock.wait();//执行睡眠

}

//干活

}

//另一个线程

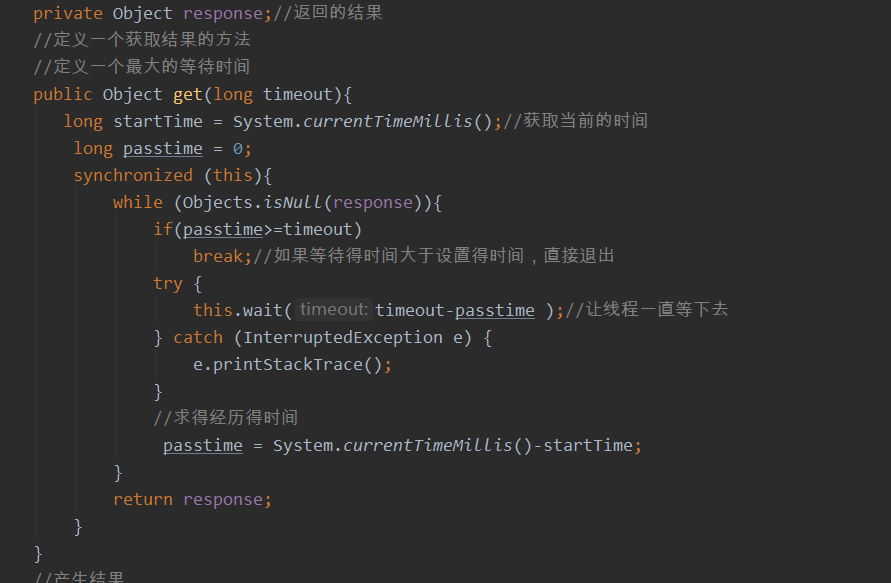
使用lock.notifyAll();将所有的线程唤醒

28.10-设计模式-保护性暂停

1、保护性暂停模式，即两个线程之间的互相依赖，一个线程必须等待另一个线程的执行完成

2、设计一个类，此类有两个互斥方法，一个是获取结果，一个是产生结果。

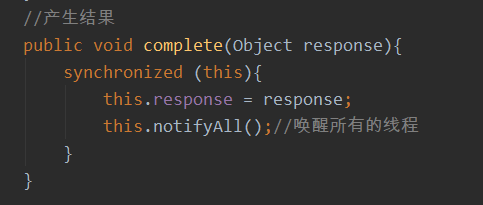
A、获取结果的方法



此方法可以设置一个最大的等待时间

等待时间为timeout-passTime ----防止虚假唤醒

B、产生结果的方法



产生完成结果,唤醒每个等待的结果的线程

C、测试方法

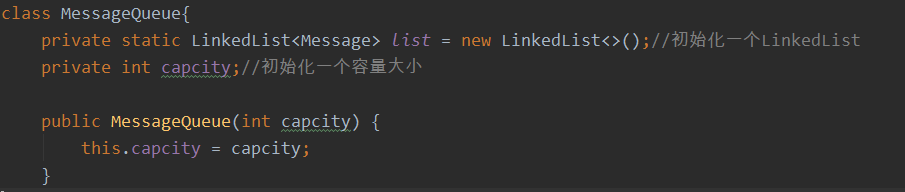


主方法，创建一个类对象和两个线程。一个线程负责获取结果，一个线程负责产生结果。

产生结果的线程，产生了结果，得到结果结果被唤醒，然后返回产生的结果的对象。

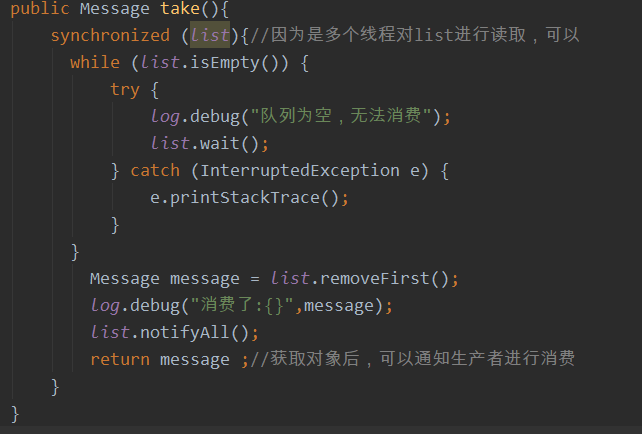
28.11-生产者消费者模式

1、消息队列（linkedList、take()方法、put()方法）



该队列包含一个双向链表的集合和一个容量大小的参数

2、take()方法



1. 因为多个线程是对同一个list进行获取插入操作，可以将其作为锁对象
2. 判断链表是否为空，为空使线程进入等待队列
3. 获取完成消息对象，唤醒生产的线程
4. put()方法



1. 判断是否链表是否达到容量的大小，达到大小，使线程进入等待状态
2. 链表中加入一个对象的时候，可以唤醒所有的消费者的线程

28.12-park和unpark方法

1、API

LockSupport.park()//使当前的线程进入等待专题

LockSupport.unpark(线程对象);//唤醒参数线程

2、与Object的wait和notify、notifyAll相比

A、wait、notify和notifyAll必须配合object Monitor使用，park、unpark不需要

B、park和unpark是以线程为单位来阻塞和唤醒线程，notify只能随机唤醒一个而notifyAll唤醒所有的线程

C、park和unpark可以先unpark，而notify和wait必须先notify

28.13-ReentrantLock

28.13.1-特点

1、可重入

可重入，同一个线程可以多次获得同一个对象锁

2、可中断（必须锁为可中断的锁，lock.lockInterrupbility）

3、可以设置超时时间

Lock.tryLock();//尝试获取锁，如果获取了，返回值为true

Lock.tryLock(1,TimeUtil.Seconds);//设置没有获取锁的等待时间

4、可以设置公平锁

5、支持多个条件变量

Lock.newCondition();//创建一个类似waitset的功能的休息室

Condition.signalAll();//叫醒多个线程

设置waitSet的类型，区分waitSet

28.14-共享模型的内存

28.14.1-可见性

Volatile-关键字，他可以避免线程从缓存中读取值，直接从主存中读取值，写入值也是直接写入到内存。

28.14.2-有序性

1、指令重排序，可以通过volatile的关键字消除指令重排序

2、synchronized保证关键字代码内不受指令的重排序的影响，但是不受synchronized的关键字的影响，不能保证是否受指令重排序的影响

3、volatile保证关键字之前执行的赋值，直接写入内存。

28.14.3-JAVA-单例模式的Double check



1. 为什么加volatile?

如果不对INSTANCE加关键字volatile，无法保证synchronized指令的重排序，创建对象时候可能会先赋值引用，再执行构造函数。如果不加volatile关键字会得到没有执行初始化的对象引用，加上volatile保证了指令不会被重排序。得到的对象引用比对是执行了构造函数初始化的工作。

1. 问题2的解答？

使用是懒汉式的模式加载对象，并且没有必要每次对所有的代码块加索。

1. 问题3的解答？

当t1线程执行对象的初始化并且返回了对象的时候，此时t2线程正好进入代码块，如果不加对象的非空判断，会再次执行构造函数的方法。

28.15-共享模型之无锁

28.15.1-什么是CAS?

1、CAS就是compareAndSet的简称，结合volatile的关键字达到原子性的效果。例如在设置值的时候会比较当前获取的值和主存的值是否有变化，如果有变化就无法执行赋值操作。

28.15.2-为什么无锁的效率会高？

因为不会发生线程的上下文切换，不会有额外的性能损耗，此外如果是单核cpu或者线程的数量比较多(线程的数量尽量小于cpu的核心数)，那么还是会获取不到cpu执行的时间片，还是会导致上下文的切换发生。

28.15.3-CAS的特点

1、Cas适合无锁，多核cpu、线程数少的场景。

2、CAS是基于乐观锁而synchronized基于悲观锁

3、CAS体现是无锁并发、无阻塞并发

A、因为没有synchronized，所以不会有阻塞不会发生上下文切换，所以效率高

B、如果竞争激烈，可以想象重试必然频繁发生，反而效率会受到影响。

28.16-线程池

28.16.1-自定义阻塞队列

1、阻塞队列的元素

A、双向链表，用于存放任务队列（阻塞队列加泛型）

B、锁对象，Reentralock，涉及到多线程从双向链表中取对象

C、条件变量-队列的等待休息室，当双向链表满了，存入等待队列中

D、条件变量-线程的等待休息室，当双向链表中没有任务，线程要进入阻塞休息室

E、容量大小，设置双向链表的大小

手机屏幕的截图

描述已自动生成

2、获取阻塞队列中的任务（含超时等待）



注:

1. awaitNanos方法返回剩余等待时间
2. signalAll方法唤醒等待休息室里的所有的任务

3、阻塞添加（含超时时间）

手机屏幕的截图

描述已自动生成

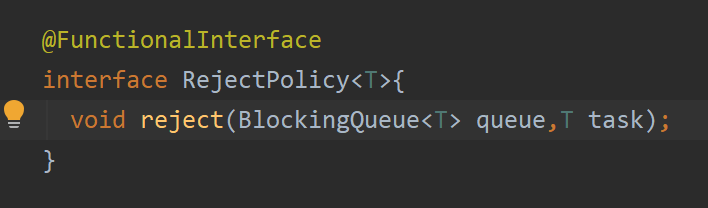
3、含有具体的拒绝策略的添加方法（使用策略模式）

手机屏幕的截图

描述已自动生成

注:

拒绝策略是定义的一个接口，通过具体的实现实现拒绝的测率



方法需要两个参数，一个是阻塞队列，一个是具体的任务对象。

有以下几种的拒绝策略:

1. 死等
2. 
3. 超时等待
4. 调用者放弃执行任务（也就具体的实现不执行任何逻辑代码）
5. 调用者抛出异常
6. 调用者自己执行（主线程执行run方法）

28.16.2-自定义线程池

1、线程池中含有的元素

A、任务队列

B、线程集合

C、核心线程数

D、获取任务的超时时间

E、时间单位

D、添加任务阻塞的拒绝策略



2、执行任务方法



注:

Woker就是一个线程对象，执行任务

判断当前线程数是否小于核心数大小，如果小于就直接交给线程执行

如果大于的话，就要加入阻塞队列

28.16.3-自定义执行线程（Worker）



注：

1. 如果传递过来的任务不为空，直接执行，如果为空，就可以去任务队列中取取任务队列，直到条件不满足退出循环，执行完任务，将task设置为null
2. 当退出循环时候，说明任务执行完成，线程可以集合中退出。

28.16.4-正确处理线程池异常

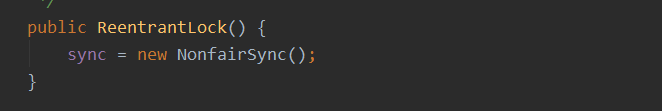
1、自己主动在代码块里执行try catch

2、使用带返回值的callable处理任务，future的get方法会输出异常的日志信息

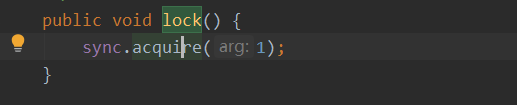
28.17-reentrantLock的工作原理

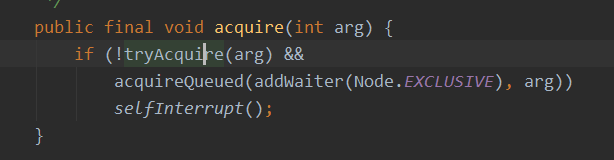
28.17.1-reentrantLock加锁成功流程

1、ReentrantLock的默认构造函数是非公平的独占锁

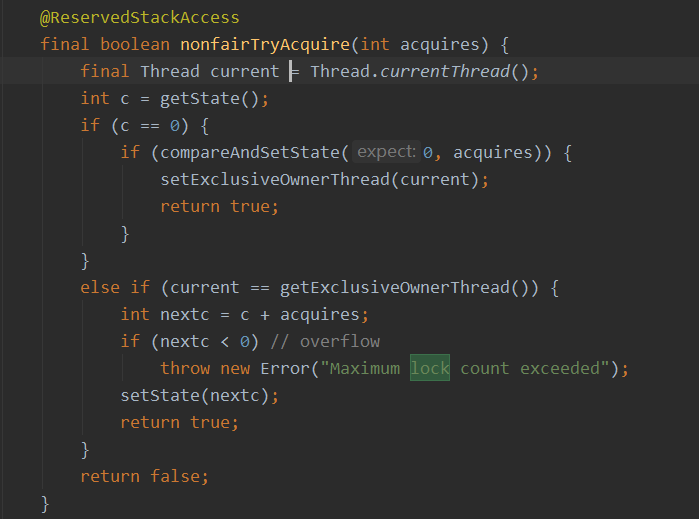


2、调用sync的lock方法





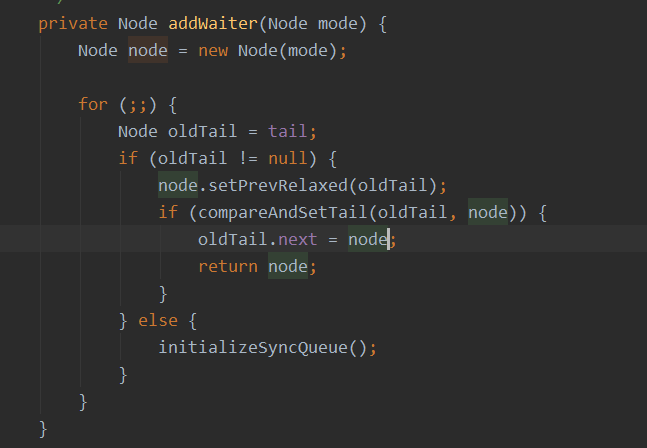
1. 分析一下tryAcquire方法



1. 该方法获取当前的state状态，判断是否等于0，如果等于0设置当前的线程为独占锁。
2. 如果线程状态不等于0，判断获取独占锁的是不是就是当前线程，如果是的话，将当前的state状态加1，这表示该线程重入的次数。
3. 除了以上两种情况，抢占锁失败，返回false，进入阻塞队列

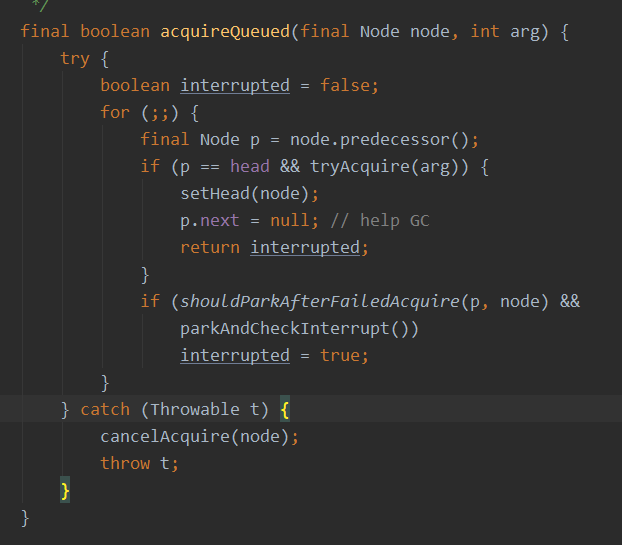
28.17.2-reentrantLock加锁失败流程

1、addWaiter-阻塞线程加入等待队列方法

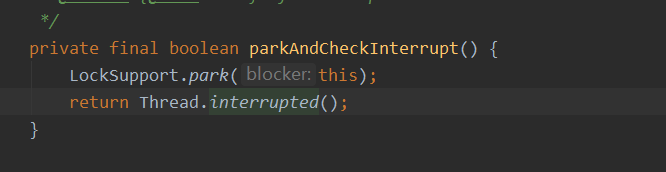


1. 初始化节点，设置关联线程的模式，独占或者共享
2. 获取尾部节点，如果尾部不为空，设置当前的线程节点为尾部节点
3. 如果尾部节点为空，说明这个双向链表为空，初始化双向链表即可。

2、acquireQueue（）方法分析



1. 首先判断当前节点的前驱节点是否头部节点，如果是，尝试获取，因为此时恰巧有其他的线程释放了锁
2. 如果获取锁不成功进入shouldParkAfterFailedAcuqire()方法
3. 上一个方法返回false再次进入parkAndCheckInterrupt方法，使用park阻塞当前线程



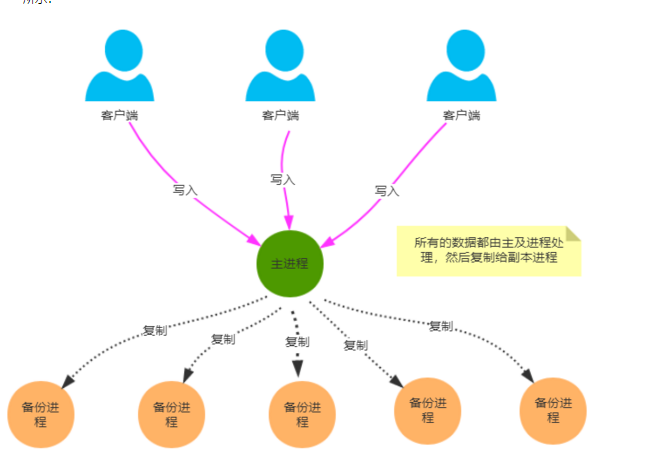
###### 29-zookeeper和dubbo

29.1-关于ZAB协议

29.1.1-ZAB协议概述

Zookeeper采用的是ZAB协议，它是一种支持崩溃恢复和原子广播协议。

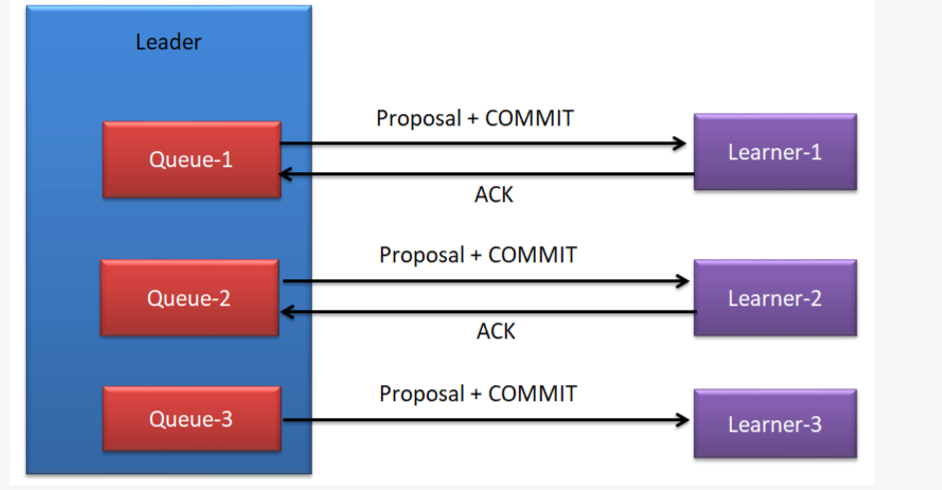
基于该协议实现了一种，主备模式。



客户端的所有请求会到主进程，然后复制到所有的备份进程中。备份进程只要超过一半以上回复ACK信息就可以进行commit提交提案。

29.1.2-初始化同步

完成Leader选举后，此时的Leader是一个准Leader，通过恢复模式下的初始同步阶段，将准Leader变为真正的Leader。



Leader为每个Lerner创建一个队列，保证有序执行事务提交。

每个Leaner执行完事务并且返回一个ACK消息，当Leader收到Learner的ACK消息中时，会将该Follwer加入到真正可用的Follwer列表中。

29.1.3-消息广播算法

1、当集群中有过半的的Follower完成了初始化状态同步，那么整个zk进入到正常的工作模式。

2、当Leader接收到事务请求的时候后，为事务分配一个全局自增的ID，即ZXID，通过ZXID可以实现事务的有序管理，将事务封装为一个提案进行执行。

3、当follwer接收到提案后会与本地的进行比较，如果大于最大的ZXID，则将提案存储到本地的日志中，并向Leader返回一个ACK消息。

4、当Leader接收到过半的ACK后，会将所有的Follwer队列发送commit消息，向所有的Observe队列发送提案。

5、当Follwer收到Commit消息的时候，会将日志中的事务更新到本地，当Observe接收到提案后，会直接将事务更新到本地。

6、zookeeper为了保证所有进程能够有序执行，只能是Leader接收请求，即使是follower接收到请求，也会转发到Leader服务器进行执行。

29.1.4-崩溃恢复

1、Leader在复制数据给所有follower后奔溃怎么办？

2、Leader在收到ACK并提交了自己，同时发送了部分commit出去崩溃。怎么办？

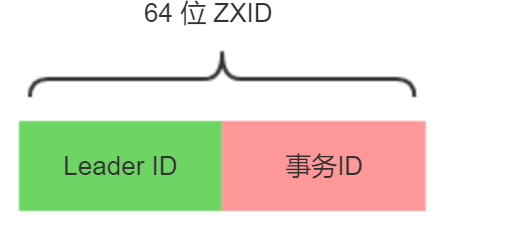
针对这些问题，ZAB定义了两个原则:

1. ZAB确保那些在Leader中提交的事务会被所有的服务器提交。
2. ZAB协议确保丢弃那些只在Leader提出复制，但没有提交的事务。

针对这个要求，Leader选举算法，保证选举出来的Leader服务器拥有集群的最大的ZXID的事务，通过初始化同步就可以恢复数据。

29.1.5-关于几个名词的解释

1、ZXID



高32位表示Leader的唯一性，低32位表示Leader中事务的唯一性，基于这样的策略，follower链接上Leader后会根据自己的ZXID和Leader的ZXID对比，要么执行事务 要么进行回滚，最终保持与Leader的数据一致性。

1. myid:是集群服务器中的唯一标识，例如:1,2,3
2. 逻辑时钟，在选举的时候称为LogicalLock，选举结束称为epoch值，epoc和logicalLock是同一个值在不同阶段的名称。
3. Zk的状态

Looking-选举状态（查找Leadder）

Following-跟随状态

Observing-观察状态

Leading-领导状态

29.1.6-Leader的选举算法

1、每个server都会投自己一票，并且将自己的投票广播出去。

2、接收投票，集群中的服务接收投票，先判断投票的有效性，如是否是本轮次投票，是否是来自looking状态的投票

3、处理投票

先检查 ZXID，ZXID大的当选领导

ZXID相等，比较myId，大的当选领导

1. 当投票超出半数的称为Leader，其他的只能投给自己。
2. 当添加新的主机进来，由于目前集群各个机器正常运行，其他只能作为follower身份加进来。
3. Leader出现宕机，那么就会重新选举，集群中的非observer会变为looking状态，重复上述的选举状态。

29.2-关于CAP原则

29.2.1-概述

1、C-一致性原则

2、A-可用性

3、P-区分容错性

29.2.2-其他

1、P的原则必须保证，而zk保证cp原则

2、zk为了保证数据的一致性，做了两个数据同步:

初始化同步-新leader产生后，follwer会将数据同步到leader中

更新同步-当Leader数据更改后，会向follwer发送广播信息，主动同步Leader中的数据

1. 不管是初始化同步还是更新同步，zk为了保证数据的唯一性，若超过半数以上的同步超时，会再次同步，此过程zk处于不可用状态。

29.3-zookeeper的应用场景

1、配置中心

2、负载均衡

3、命名服务(RPC框架的命名服务)

4、DNS服务

5、集群管理

6、分布式锁

29.3.1-关于分布式锁

###### 30-关于JVM

30.1-内存结构

30.1.1-程序计数器

1、作用-记住下一条指令的地址

2、特点:

A、是线程私有的

B、不存在内存溢出

30.1.2-虚拟机栈

1、栈-线程运行需要的内存空间

2、栈帧-是每个方法运行时需要的内存空间，一个栈可能有多个栈帧，比如在一个方法里面调用了其他的方法

3、栈帧里包含了方法运行的参数、局部变量、返回值等

4、每个线程对应一个活动栈帧，对应着当前正在执行的方法

5、问题辨析:

A、垃圾回收是否涉及到栈内存

不涉及，因为栈内存随着方法的执行完成，自己主动释放

B、栈内存分配越大越好吗？

不是，栈内存分配的大，相当于运行的线程数量会减少。每个线程对应的内存增加，而

总的内存是固定的，所以导致运行的线程数量减少。

C、方法内的局部变量是否线程安全？

..如果方法内的局部变量没有逃离方法的作用范围，那么就是安全的

..如果是局部变量引用了对象，并逃离了方法的作用范围，需要考虑线程的安全

30.1.2.1-栈内存溢出

1、栈帧过多导致内存溢出（无限递归调用）

Java会抛出StackOverFlower的异常

2、栈帧太大导致（此种情况较少）

30.1.2.2-线程运行诊断

1、cpu占用时间过长

定位:

1. 用top命令定位哪个线程占用cpu比较多
2. Ps H -eo pid,tid,%cpu grep | 进程id（进一步确认哪个线程占用的时间比较高）
3. jstack进程ID,可以根据线程ID定位到哪一行的代码出现问题。

2、案例2：运行很长时间没有得到结果

可能原因是由于线程之间的死锁导致的

30.1.3-本地方法栈（Native Method Stack）

底层是C++语言编写的方法，执行时候的内存区域。

30.1.4-堆

1、通过new方式创建的对象，都会在堆内存中产生

2、特点

A、它是线程共享的，要考虑到线程的安全性问题

B、有垃圾回收的机制存在

30.1.4.1-堆内存泄露

1、java会抛出java.land.OutOfMerroyError

可以设置虚拟机启动参数 -Xmx8m（8M）的内存空间，来设置堆的内存大小空间

30.1.4.2-堆内存诊断

1、jps工具

查看系统当前的进程ID

2、jmap -heap 进程ID

查看堆内存的占用情况

3、jconsole图形工具

输入jconsole-命令打开图形工具

4、案例:执行垃圾回收，内存占用还是很高

使用jvisualvm—打开jvm的图形检测工具，可以使用heap dump工具来检测堆内存快照信息.

30.1.5-方法区

30.1.5.1-方法区一些定义

1、组成部分

A、1.6之前忽略

B、1.8-元空间-class、classloader、常量池

2、常量池

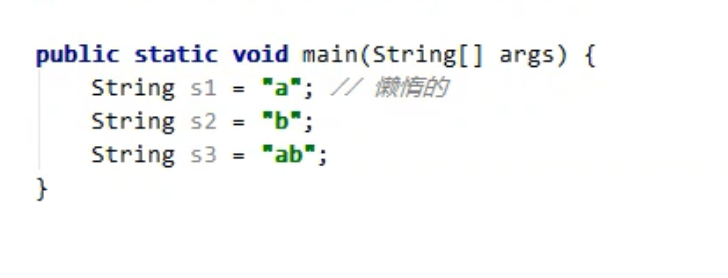
常量池就是一张表，虚拟机根据这张表找到要执行的类名、方法名、参数类型、字面量等信息

3、运行时常量

常量池是 \*.class 文件中的，当该类被加载，它的常量池信息就会放入运行时常量

池，并把里面的符号地址变为真实地址

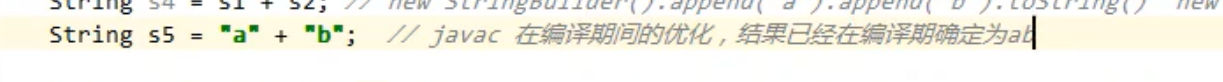
30.1.5.2-StringTable串池和字符串常量的关系



1. 常量池的信息，都会被加载到运行时的常量池中，这时的 a，b，ab都还是符号，还不是java对象
2. 运行到String s1 = a;会把a符号变为java对象，并a的java字符串对象放入串池（数据结构是一个hash表）
3. 同理加载 b和ab（是一种懒惰加载的特性）
4. 先到串池找，如果串池中找，就可以取串池的对象，每个对象是唯一的



1. 当执行s1+s2=ab的时候虽然和s3的结果一样，但是s3是在常量池里，而s4是通过构造一个新的string对象，所以两个对象还是不同的。变量的字符串拼接是通过stringbuilder进行拼接。



Javac会在编译器进行优化，结果确定为ab直接在常量池里寻找

所以结果是s5等于s3。常量的拼接是常量池中直接拼接。

7、可以使用intern方法主动将字符串对象放入到常量池中

30.15.3-StringTable性能调优

1、调整-XX:StringTableSize=桶个数

2、如果相同的字符串对象很多，可以调用intern方法，将字符串对象入常量池，那些重复对象由于没有被引用，会被GC垃圾回收。

30.1.6-直接内存

30.1.6.1-直接内存的概述

1、常用于NIO操作时，充当数据的缓冲区

2、分配回收成本较高，但读写性能高

3、不受JVM的内存回收管理

30.1.6.2-直接内存的分配和回收原理

1、使用了unsafe对象对直接内存的分配和回收，并且回收要主动调用freeMemory方法

2、ByteBuffer的实现内部，使用了Cleaner（虚拟引用）来监测ByteBuffer，一旦ByteBuffer对象被回收了，那么就由ReferenceHandler线程通过Cleaner的clean方法调用freeMerrory方法来释放直接内存。

30.2-垃圾回收

30.2.1-如何判断对象是否可以回收

1、java虚拟机中的垃圾回收算法是可达性算法来探索所有存活对象

2、扫描堆中的对象，看是否能够沿着GC\_ROOT对象为起点的引用链找到该对象，找不到表示可以回收。

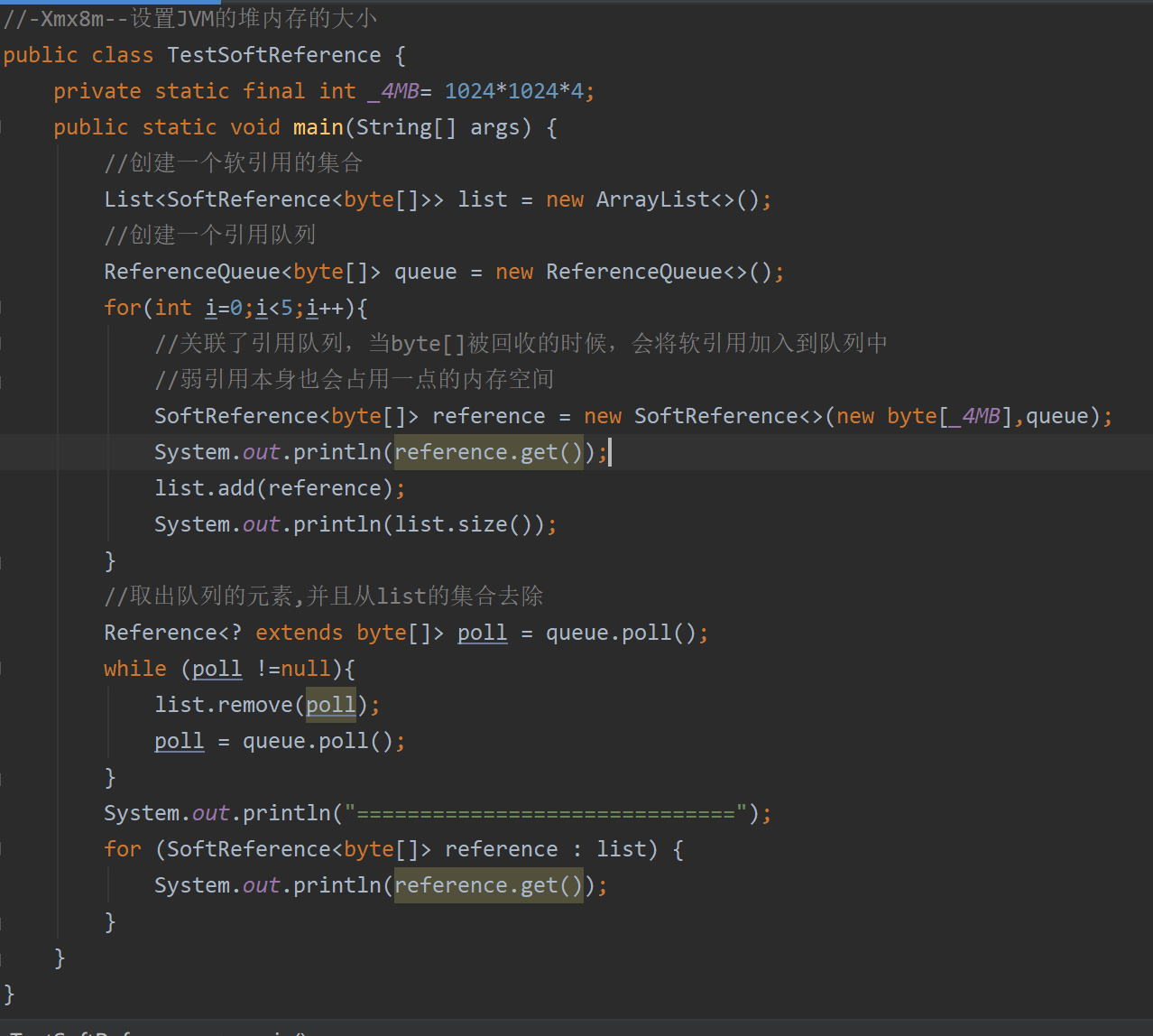
3、哪些对象可以作为GC\_ROOT？

30.2.2-四种引用

1、强引用

程序代码直接赋值的引用类型。

2、软引用



3、弱引用

4、虚引用

5、终结器引用

30.2.3-垃圾回收算法

30.2.3.1-标记清除

1、整理可以被垃圾回收的对象

2、清除对象的地址空间，将对象的起始地址到结束地址标记为空闲区域

3、优点:速度快;缺点:容易产生内存碎片

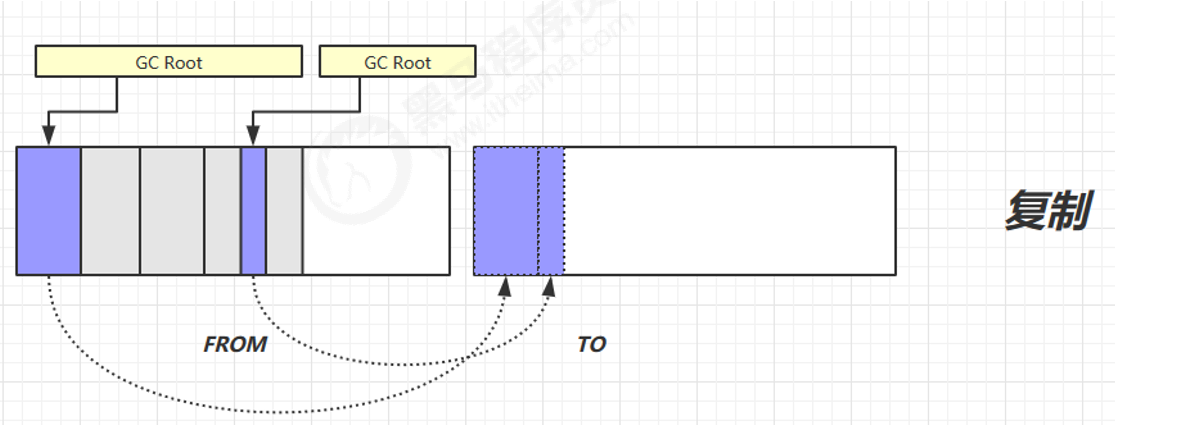
30.2.3.2-标记整理

1、先标记可以回收的垃圾对象

2、整理出空闲的内存地址为一片连续的内存地址

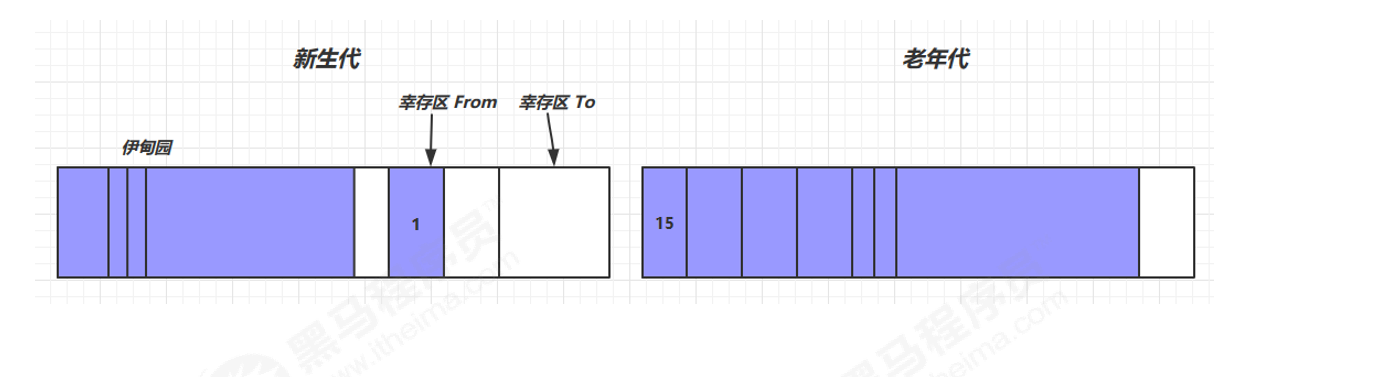
3、优点:少量的内存碎片;缺点:时间较长

30.2.3.3-复制算法



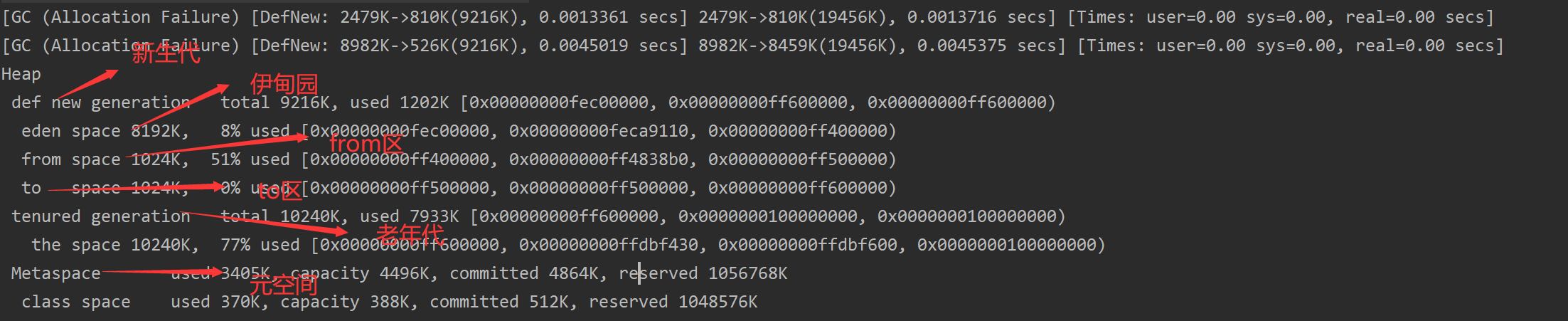
1. 将存活下来的对象负责到TO的区域
2. 将from区域下的内存进行清除为空闲地址
3. FROM 和TO进行调换

30.2.3.4-分代垃圾回收



1. 对象首先分配在伊甸园区域
2. 新生代的空间不足的时候，会触发minor gc，伊甸园和from区域中存活的对象会复制到to中，存活的对象加1并且交换from to
3. Minor gc 会引发STW（stop the world），暂停其他的用户线程，等垃圾回收结束，才恢复运行
4. 当对象的寿命超过阈值，会晋升到老年代（寿命大概是15）
5. 当老年代的空间不足的时候，会尝试触发一次minor gc，如果之后空间仍不足，就会触发full gc，同时STW的时间会更长。

30.2.3.5-GC分析



1. 当新生代的内存不足以存放数据的时候，会直接GC到老年代的内存中

30.2.3.4-垃圾回收器

30.2.3.5-垃圾回收调优

30.2.3.5.3-最快GC

1、答案是不发生GC

2、查看FullGC前后占用的内存，要考虑一下几个问题

A、数据是不是太多

Select \* from 表 limit n（要加分页查询）

B、数据表示是否太臃肿

对象图、对象大小 Integer 16 int 4

C、是否存在内存泄露

Staic Map map =

软

弱

第三方的实现缓存

30.2.3.5.4-新生代调优

1、为啥优先做新生代调优？

A、所有的new操作分配的对象非常廉价

B、死亡对象的回收代价是零

C、大部分对象用过即死

D、MinorGc的时间远远低于FullGC的时间

2、越大越好吗？

A、新生代能容纳所有[并发量\*（请求+响应）]数据