第 4 天 java 高级特性增强

今天内容安排：
1、掌握多线程
2、掌握并发包下的队列
3、了解 JMS
4、掌握 JVM 技术
5、掌握反射和动态代理

➢ java 多线程增强

.1. java 多线程基本知识

.1.1. 进程介绍

不管是我们开发的应用程序，还是我们运行的其他的应用程序，都需要先把程序安装在本地的硬盘上。然后找到这个程序的启动文件，启动程序的时候，其实是电脑把当前的这个程序加载到内存中，在内存中需要给当前的程序分配一段独立的运行空间。这片空间就专门负责当前这个程序的运行。

不同的应用程序运行的过程中都需要在内存中分配自己独立的运行空间，彼此之间不会相互的影响。我们把每个独立应用程序在内存的独立空间称为当前应用程序运行的一个进程。

进程：它是内存中的一段独立的空间，可以负责当前应用程序的运行。当前这个进程负责调度当前程序中的所有运行细节。

.1.2. 线程介绍

启动的 QQ 聊天软件，需要和多个人进行聊天。这时多个人之间是不能相互影响，但是它们都位于当前 QQ 这个软件运行时所分配的内存的独立空间中。

在一个进程中，每个独立的功能都需要独立的去运行，这时又需要把当前这个进程划分成多个运行区域，每个独立的小区域（小单元）称为一个线程。

线程：它是位于进程中，负责当前进程中的某个具备独立运行资格的空间。

进程是负责整个程序的运行，而线程是程序中具体的某个独立功能的运行。一个进程中至少应该有一个线程。

.1.3. 多线程介绍

现在的操作系统基本都是多用户，多任务的操作系统。每个任务就是一个进程。而在这个进程中就会有线程。

真正可以完成程序运行和功能的实现靠的是进程中的线程。

多线程：在一个进程中，我们同时开启多个线程，让多个线程同时去完成某些任务（功能）。(比如后台服务系统，就可以用多个线程同时响应多个客户的请求)

多线程的目的：提高程序的运行效率。

.1.4. 多线程运行的原理

cpu 在线程中做时间片的切换。

其实真正电脑中的程序的运行不是同时在运行的。CPU 负责程序的运行，而 CPU 在运行程序的过程中某个时刻点上，它其实只能运行一个程序。而不是多个程序。而 CPU 它可以在多个程序之间进行高速的切换。而切换频率和速度太快，导致人的肉眼看不到。每个程序就是进程， 而每个进程中会有多个线程，而 CPU 是在这些线程之间进行切换。了解了 CPU 对一个任务的执行过程，我们就必须知道，多线程可以提高程序的运行效率，但不能无限制的开线程。

.1.5. 实现线程的两种方式

1、继承 Thread 的方式

见代码 MyThreadWithExtends

2、声明实现 Runnable 接口的方式

见代码 MyThreadWithImpliment

.2. java 同步关键词解释

.2.1. synchronized

加同步格式：

```
synchronized( 需要一个任意的对象（锁） ){
    代码块中放操作共享数据的代码。
}
```

见代码 MySynchronized

- synchronized 的缺陷

synchronized 是 java 中的一个关键字，也就是说是 Java 语言内置的特性。

如果一个代码块被 synchronized 修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2）线程执行发生异常，此时 JVM 会让线程自动释放锁。

例子 1：

如果这个获取锁的线程由于要等待 IO 或者其他原因（比如调用 sleep 方法）被阻塞了，但是又没有释放锁，其他线程便只能干巴巴地等待，试想一下，这多么影响程序执行效率。

因此就需要有一种机制可以不让等待的线程一直无期限地等待下去(比如只等待一定的时间或者能够响应中断），通过 Lock 就可以办到。

例子 2：

当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。

但是采用 synchronized 关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过 Lock 就可以办到。

另外，通过 Lock 可以知道线程有没有成功获取到锁。这个是 synchronized 无法办到的。

总的来说，也就是说 Lock 提供了比 synchronized 更多的功能。

2.2 lock

➢ lock 和 synchronized 的区别

1）Lock 不是 Java 语言内置的，synchronized 是 Java 语言的关键字，因此是内置特性。Lock 是一个类，通过这个类可以实现同步访问；

2）Lock 和 synchronized 有一点非常大的不同，采用 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。

➢ java.util.concurrent.locks 包下常用的类

✧ Lock

首先要说明的就是 Lock，通过查看 Lock 的源码可知，Lock 是一个接口：

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    void unlock();
    }
```

Lock 接口中每个方法的使用：

lock()、tryLock()、tryLock(long time, TimeUnit unit)、lockInterruptibly()是用来获取锁的。

unLock()方法是用来释放锁的。

四个获取锁方法的区别：

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。

由于在前面讲到如果采用 Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此一般来说，使用 Lock 必须在 try{}catch{}块中进行，并且将释放锁的操作放在 finally 块中进行，以保证锁一定被被释放，防止死锁的发生。

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，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。

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。如果如果一开始拿到锁或者在等待期间内拿到了锁，则返回 true。

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。也就使说，当两个线程同时通过 lock.lockInterruptibly()想获取某个锁时，假若此时线程 A 获取到了锁，而线程 B 只有等待，那么对线程 B 调用 threadB.interrupt()方法能够中断线程 B 的等待过程。

注意，当一个线程获取了锁之后，是不会被 interrupt()方法中断的。

因此当通过 lockInterruptibly()方法获取某个锁时，如果不能获取到，只有进行等待的情况下，是可以响应中断的。

而用 synchronized 修饰的话，当一个线程处于等待某个锁的状态，是无法被中断的，只有一直等待下去。

✧ ReentrantLock

直接使用 lock 接口的话，我们需要实现很多方法，不太方便，ReentrantLock 是唯一实现了 Lock 接口的类，并且 ReentrantLock 提供了更多的方法，ReentrantLock，意思是“可重入锁”。

以下是 ReentrantLock 的使用案例：

例子 1，lock()的正确使用方法
见代码 MyLockTest

例子 2，tryLock()的使用方法
见代码 MyTryLock

例子 3，lockInterruptibly()响应中断的使用方法：
见代码 MyInterruptibly

✧ ReadWriteLock

ReadWriteLock 也是一个接口，在它里面只定义了两个方法：

```
public interface ReadWriteLock {
    /**
     * Returns the lock used for reading.
     *
     * @return the lock used for reading.
     */
    Lock readLock();

    /**
     * Returns the lock used for writing.
     *
     * @return the lock used for writing.
     */
    Lock writeLock();
}
```

一个用来获取读锁，一个用来获取写锁。也就是说将文件的读写操作分开，分成 2 个锁来分配给线程，从而使得多个线程可以同时进行读操作。下面的 ReentrantReadWriteLock 实现了 ReadWriteLock 接口。

✧ ReentrantReadWriteLock

ReentrantReadWriteLock 里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和 writeLock()用来获取读锁和写锁。

下面通过几个例子来看一下 ReentrantReadWriteLock 具体用法。

例子 1：假如有多个线程要同时进行读操作的话，先看一下 synchronized 达到的效果

见代码 MySynchronizedReadWrite

例子 2：改成用读写锁的话：

见代码 MyReentrantReadWriteLock

注意：

不过要注意的是，如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。

如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁。

✧ Lock 和 synchronized 的选择

1）Lock 是一个接口，而 synchronized 是 Java 中的关键字，synchronized 是内置的语言实现；

2）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；

3）Lock 可以让等待锁的线程响应中断，而 synchronized 却不行，使用 synchronized 时，等待的线程会一直等待下去，不能够响应中断；

4）通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

5）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。所以说，在具体使用时要根据适当情况选择。

➢ java 并发包

.1. java 并发包介绍

JDK5.0 以后的版本都引入了高级并发特性，大多数的特性在 java.util.concurrent 包中，是专门用于多线程发编程的，充分利用了现代多处理器和多核心系统的功能以编写大规模并发应用程序。主要包含**原子量、并发集合、同步器**、可重入锁，并对线程池的构造提供了强力的支持。

线程池

✧ 线程池的 5 中创建方式：

1、 Single Thread Executor：只有一个线程的线程池，因此所有提交的任务是顺序执行，
代码： Executors.newSingleThreadExecutor()

2、 Cached Thread Pool：线程池里有很多线程需要同时执行，老的可用线程将被新的任务触发重新执行，如果线程超过 60 秒内没执行，那么将被终止并从池中删除，
代码：Executors.newCachedThreadPool()

3、 Fixed Thread Pool：拥有固定线程数的线程池，如果没有任务执行，那么线程会一直等待，
代码： Executors.newFixedThreadPool(4)
在构造函数中的参数 4 是线程池的大小，你可以随意设置，也可以和 cpu 的核数量保持一致，获取 cpu 的核数量 int cpuNums = Runtime.getRuntime().availableProcessors();

4、 Scheduled Thread Pool：用来调度即将执行的任务的线程池，可能是不是直接执行，每隔多久执行一次... 策略型的
代码：Executors.newScheduledThreadPool()

5、 Single Thread Scheduled Pool：只有一个线程，用来调度任务在指定时间执行，代码：Executors.newSingleThreadScheduledExecutor()

✧ 线程池的使用

提交 Runnable ，任务完成后 Future 对象返回 null
调用 excute,提交任务，匿名 Runable 重写 run 方法, run 方法里是业务逻辑
见代码：ThreadPoolWithRunable

提交 Callable，该方法返回一个 Future 实例表示任务的状态
调用 submit 提交任务，匿名 Callable,重写 call 方法，有返回值，获取返回值会阻塞,一直要等到线程任务返回结果
见代码：ThreadPoolWithcallable

.2. java 并发包消息队列及在开源软件中的应用

BlockingQueue 也是 java.util.concurrent 下的主要用来控制线程同步的工具。
主要的方法是：put、take 一对阻塞存取；add、poll 一对非阻塞存取。

插入:

1)add(anObject):把 anObject 加到 BlockingQueue 里,即如果 BlockingQueue 可以容纳,则返回 true,否则抛出异常,不好

2)offer(anObject):表示如果可能的话,将 anObject 加到 BlockingQueue 里,即如果 BlockingQueue 可以容纳,则返回 true,否则返回 false.

3)put(anObject):把 anObject 加到 BlockingQueue 里,如果 BlockQueue 没有空间,则调用此方法的线程被阻断直到 BlockingQueue 里面有空间再继续，有阻塞，放不进去就等待

读取：

4)poll(time):取走 BlockingQueue 里排在首位的对象,若不能立即取出,则可以等 time 参数规定的时间,取不到时返回 null；取不到返回 null

5)take():取走 BlockingQueue 里排在首位的对象,若 BlockingQueue 为空,阻断进入等待状态直到 Blocking 有新的对象被加入为止；阻塞，取不到就一直等

其他

int remainingCapacity();返回队列剩余的容量，在队列插入和获取的时候，不要瞎搞，数据可能不准，不能保证数据的准确性

boolean remove(Object o); 从队列移除元素，如果存在，即移除一个或者更多，队列改变了返回 true

public boolean contains(Object o); 查看队列是否存在这个元素，存在返回 true

int drainTo(Collection<? super E> c); //移除此队列中所有可用的元素,并将它们添加到给定 collection 中。取出放到集合中

int drainTo(Collection<? super E> c, int maxElements); 和上面方法的区别在于，指定了移动的数量; 取出指定个数放到集合

BlockingQueue 有四个具体的实现类,常用的两种实现类为：

1、ArrayBlockingQueue：一个由数组支持的有界阻塞队列，规定大小的 BlockingQueue,其构造函数必须带一个 int 参数来指明其大小.其所含的对象是以 FIFO(先入先出)顺序排序的。

2、LinkedBlockingQueue：大小不定的 BlockingQueue,若其构造函数带一个规定大小的参数,生成的 BlockingQueue 有大小限制,若不带大小参数,所生成的 BlockingQueue 的大小由 Integer.MAX_VALUE 来决定.其所含的对象是以 FIFO(先入先出)顺序排序的。

LinkedBlockingQueue 可以指定容量，也可以不指定，不指定的话，默认最大是 Integer.MAX_VALUE,其中主要用到 put 和 take 方法，put 方法在队列满的时候会阻塞直到有队列成员被消费，take 方法在队列空的时候会阻塞，直到有队列成员被放进来。

LinkedBlockingQueue 和 ArrayBlockingQueue 区别：

LinkedBlockingQueue 和 ArrayBlockingQueue 比较起来,它们背后所用的数据结构不一样,导致 LinkedBlockingQueue 的数据吞吐量要大于 ArrayBlockingQueue,但在线程数量很大时其性能的可预见性低于 ArrayBlockingQueue.

生产者消费者的示例代码：
见代码

➢ java 并发编程的一些总结

.1. 不应用线程池的缺点

有些开发者图省事，遇到需要多线程处理的地方，直接 new Thread(...).start()，对于一般场景是没问题的，但如果是在并发请求很高的情况下，就会有些隐患：

- 新建线程的开销。线程虽然比进程要轻量许多，但对于 JVM 来说，新建一个线程的代价还是挺大的，决不同于新建一个对象
- 资源消耗量。没有一个池来限制线程的数量，会导致线程的数量直接取决于应用的并发量，这样有潜在的线程数据巨大的可能，那么资源消耗量将是巨大的
- 稳定性。当线程数量超过系统资源所能承受的程度，稳定性就会成问题

.2. 制定执行策略

在每个需要多线程处理的地方，不管并发量有多大，需要考虑线程的执行策略

- 任务以什么顺序执行
- 可以有多少个任务并发执行
- 可以有多少个任务进入等待执行队列
- 系统过载的时候，应该放弃哪些任务？如何通知到应用程序？
- 一个任务的执行前后应该做什么处理

.3. 线程池的类型

不管是通过 Executors 创建线程池，还是通过 Spring 来管理，都得清楚知道有哪几种线程池：

- FixedThreadPool：定长线程池，提交任务时创建线程，直到池的最大容量，如果有线程非预期结束，会补充新线程
- CachedThreadPool：可变线程池，它犹如一个弹簧，如果没有任务需求时，它回收空闲线程，如果需求增加，则按需增加线程，不对池的大小做限制
- SingleThreadExecutor：单线程。处理不过来的任务会进入 FIFO 队列等待执行
- SecheduledThreadPool：周期性线程池。支持执行周期性线程任务

其实，这些不同类型的线程池都是通过构建一个 ThreadPoolExecutor 来完成的，所不同的是 corePoolSize,maximumPoolSize,keepAliveTime,unit,workQueue,threadFactory 这么几个参数。具体可以参见 JDK DOC。

.4. 线程池饱和策略

由以上线程池类型可知，除了 CachedThreadPool 其他线程池都有饱和的可能，当饱和以后就需要相应的策略处理请求线程的任务，比如，达到上限时通过 ThreadPoolExecutor.setRejectedExecutionHandler 方法设置一个拒绝任务的策略，JDK 提供了 AbortPolicy、CallerRunsPolicy、DiscardPolicy、DiscardOldestPolicy 几种策略，具体差异可见 JDK DOC

.5. 线程无依赖性

多线程任务设计上尽量使得各任务是独立无依赖的，所谓依赖性可两个方面：

- 线程之间的依赖性。如果线程有依赖可能会造成死锁或饥饿
- 调用者与线程的依赖性。调用者得监视线程的完成情况，影响可并发量

当然，在有些业务里确实需要一定的依赖性，比如调用者需要得到线程完成后结果，传统的 Thread 是不便完成的，因为 run 方法无返回值，只能通过一些共享的变量来传递结果，但在 Executor 框架里可以通过 Future 和 Callable 实现需要有返回值的任务，当然线程的异步性导致需要有相应机制来保证调用者能等待任务完成，关于 Future 和 Callable 的用法前文已讲解；

➢ java JMS 技术

.1. 什么是 JMS

JMS 即 Java 消息服务（Java Message Service）应用程序接口是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。Java 消息服务是一个与具体平台无关的 API，绝大多数 MOM 提供商都对 JMS 提供支持。

JMS 是一种与厂商无关的 API，用来访问消息收发系统消息。它类似于 JDBC(Java Database Connectivity)：这里，JDBC 是可以用来访问许多不同关系数据库的 API，而 JMS 则提供同样与厂商无关的访问方法，以访问消息收发服务。许多厂商都支持 JMS，包括 IBM 的 MQSeries、BEA 的 Weblogic JMS service 和 Progress 的 SonicMQ，这只是几个例子。JMS 使您能够通过消息收发服务（有时称为消息中介程序或路由器）从一个 JMS 客户机向另一个 JMS 客户机发送消息。消息是 JMS 中的一种类型对象，由两部分组成：报头和消息主体。报头由路由信息以及有关该消息的元数据组成。消息主体则携带着应用程序的数据或有效负载。根据有效负载的类型来划分，可以将消息分为几种类型，它们分别携带：简单文本(TextMessage)、可序列化的对象 (ObjectMessage)、属性集合 (MapMessage)、字节流(BytesMessage)、原始值流 (StreamMessage)，还有无有效负载的消息 (Message)。

.2. JMS 规范

.2.1. 专业技术规范

JMS（Java Messaging Service）是 Java 平台上有关面向消息中间件(MOM)的技术规范，它便于消息系统中的 Java 应用程序进行消息交换,并且通过提供标准的产生、发送、接收消息的接口简化企业应用的开发，翻译为 Java 消息服务。

.2.2. 体系架构

JMS 由以下元素组成。
JMS 提供者 provider：连接面向消息中间件的，JMS 接口的一个实现。提供者可以是 Java 平台的 JMS 实现，也可以是非 Java 平台的面向消息中间件的适配器。
JMS 客户：生产或消费基于消息的 Java 的应用程序或对象。
JMS 生产者：创建并发送消息的 JMS 客户。
JMS 消费者：接收消息的 JMS 客户。
JMS 消息：包括可以在 JMS 客户之间传递的数据的对象
JMS 队列：一个容纳那些被发送的等待阅读的消息的区域。与队列名字所暗示的意思不同，

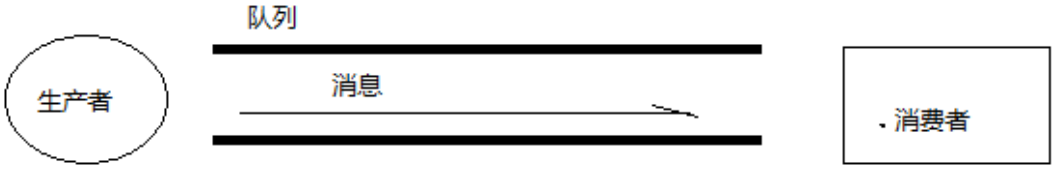
消息的接受顺序并不一定要与消息的发送顺序相同。一旦一个消息被阅读，该消息将被从队列中移走。

JMS 主题：一种支持发送消息给多个订阅者的机制。

.2.3. Java 消息服务应用程序结构支持两种模型

1、 点对点或队列模型

在点对点或队列模型下，一个生产者向一个特定的队列发布消息，一个消费者从该队列中读取消息。这里，生产者知道消费者的队列，并直接将消息发送到消费者的队列。



这种模式被概括为：

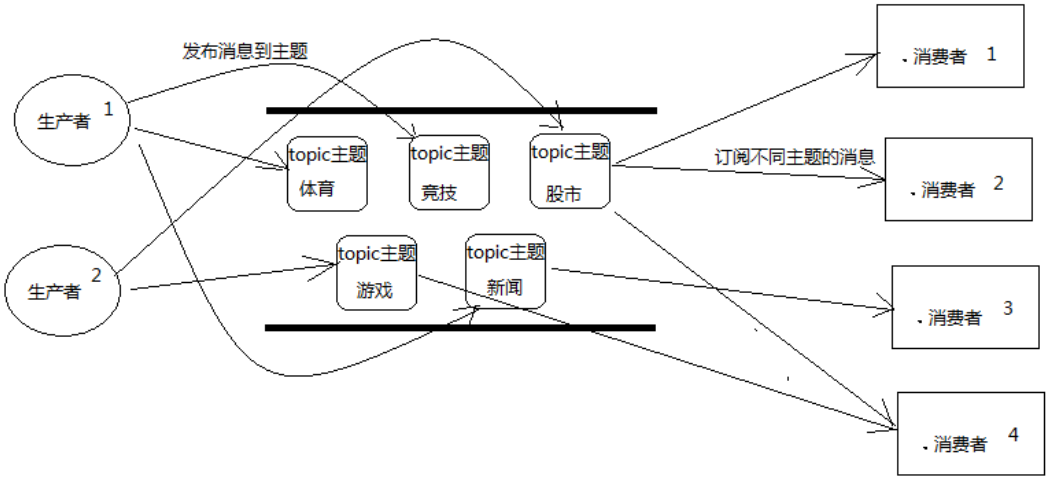
只有一个消费者将获得消息

生产者不需要在接收者消费该消息期间处于运行状态，接收者也同样不需要在消息发送时处于运行状态。

每一个成功处理的消息都由接收者签收

2、发布者/订阅者模型

发布者/订阅者模型支持向一个特定的消息主题发布消息。0 或多个订阅者可能对接收来自特定消息主题的消息感兴趣。在这种模型下，发布者和订阅者彼此不知道对方。这种模式好比是匿名公告板。



这种模式被概括为：

多个消费者可以获得消息

在发布者和订阅者之间存在时间依赖性。发布者需要建立一个订阅（subscription），以便客户能够订阅。订阅者必须保持持续的活动状态以接收消息，除非订阅者建立了持久的订阅。在那种情况下，在订阅者未连接时发布的消息将在订阅者重新连接时重新发布。

.2.4. 代码演示

1.下载 ActiveMQ
去官方网站下载：http://activemq.apache.org/

2.运行 ActiveMQ
解压缩 apache-activemq-5.5.1-bin.zip，
修改配置文件 activeMQ.xml，将 0.0.0.0 修改为 localhost

```
<transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61616"/>
        <transportConnector name="ssl"         uri="ssl://localhost:61617"/>
        <transportConnector name="stomp"      uri="stomp://localhost:61613"/>
       <transportConnector uri="http://localhost:8081"/>
        <transportConnector uri="udp://localhost:61618"/>
```

然后双击 apache-activemq-5.5.1\bin\activemq.bat 运行 ActiveMQ 程序。
启动 ActiveMQ 以后，登陆：http://localhost:8161/admin/，创建一个 Queue，命名为 FirstQueue。

3.运行代码

.2.5. 常用的 JMS 实现

要使用 Java 消息服务，你必须要有一个 JMS 提供者，管理会话和队列。既有开源的提供者也有专有的提供者。
开源的提供者包括：
Apache ActiveMQ
JBoss 社区所研发的 HornetQ
Joram
Coridan 的 MantaRay
The OpenJMS Group 的 OpenJMS
专有的提供者包括：
BEA 的 BEA WebLogic Server JMS
TIBCO Software 的 EMS
GigaSpaces Technologies 的 GigaSpaces
Softwired 2006 的 iBus
IONA Technologies 的 IONA JMS
SeeBeyond 的 IQManager（2005 年 8 月被 Sun Microsystems 并购）
webMethods 的 JMS+ -
my-channels 的 Nirvana
Sonic Software 的 SonicMQ
SwiftMQ 的 SwiftMQ
IBM 的 WebSphere MQ

➢ java 动态代理、反射

.1.1. 反射

通过反射的方式可以获取 class 对象中的属性、方法、构造函数等，一下是实例：

```
package cn.java.reflect;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

import org.junit.Before;
import org.junit.Test;

public class MyReflect {
	public String className = null;
	@SuppressWarnings("rawtypes")
	public Class personClass = null;
	/**
	 * 反射 Person 类
	 * @throws Exception
	 */
	@Before
	public void init() throws Exception {
		className = "cn.java.reflect.Person";
		personClass = Class.forName(className);
	}
	/**
	 *获取某个 class 文件对象
	 */
	@Test
	public void getClassName() throws Exception {
		System.out.println(personClass);
	}
	/**
	 *获取某个 class 文件对象的另一种方式
	 */
	@Test
	public void getClassName2() throws Exception {
```

```
        System.out.println(Person.class);
    }
    /**
     *创建一个 class 文件表示的真实对象，底层会调用空参数的构造方法
     */
    @Test
    public void getNewInstance() throws Exception {
        System.out.println(personClass.newInstance());
    }
    /**
     *获取非私有的构造函数
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    @Test
    public void getPublicConstructor() throws Exception {
        Constructor    constructor    = personClass.getConstructor(Long.class,String.class);
        Person person = (Person)constructor.newInstance(100L,"zhangsan");
        System.out.println(person.getId());
        System.out.println(person.getName());
    }
    /**
     *获得私有的构造函数
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    @Test
    public void getPrivateConstructor() throws Exception {
        Constructor con = personClass.getDeclaredConstructor(String.class);
        con.setAccessible(true);//强制取消 Java 的权限检测
        Person person2 = (Person)con.newInstance("zhangsan");
        System.out.println(person2.getName());
    }
    /**
     *获取非私有的成员变量
     */
    @SuppressWarnings({ "rawtypes", "unchecked" })
    @Test
    public void getNotPrivateField() throws Exception {
        Constructor    constructor    = personClass.getConstructor(Long.class,String.class);
        Object obj = constructor.newInstance(100L,"zhangsan");

        Field field = personClass.getField("name");
        field.set(obj, "lisi");
        System.out.println(field.get(obj));
    }
```

```
/**
 *获取私有的成员变量
 */
@SuppressWarnings({ "rawtypes", "unchecked" })
@Test
public void getPrivateField() throws Exception {
	Constructor	constructor	= personClass.getConstructor(Long.class);
	Object obj = constructor.newInstance(100L);

	Field field2 = personClass.getDeclaredField("id");
	field2.setAccessible(true);//强制取消 Java 的权限检测
	field2.set(obj,10000L);
	System.out.println(field2.get(obj));
}
/**
 *获取非私有的成员函数
 */
@SuppressWarnings({ "unchecked" })
@Test
public void getNotPrivateMethod() throws Exception {
	System.out.println(personClass.getMethod("toString"));

	Object obj = personClass.newInstance();//获取空参的构造函数
	Object object = personClass.getMethod("toString").invoke(obj);
	System.out.println(object);
}
/**
 *获取私有的成员函数
 */
@SuppressWarnings("unchecked")
@Test
public void getPrivateMethod() throws Exception {
	Object obj = personClass.newInstance();//获取空参的构造函数
	Method method = personClass.getDeclaredMethod("getSomeThing");
	method.setAccessible(true);
	Object value = method.invoke(obj);
	System.out.println(value);

}
/**
 *
 */
@Test
public void otherMethod() throws Exception {
```

```
        //当前加载这个 class 文件的那个类加载器对象
        System.out.println(personClass.getClassLoader());
        //获取某个类实现的所有接口
        Class[] interfaces = personClass.getInterfaces();
        for (Class class1 : interfaces) {
            System.out.println(class1);
        }
        //反射当前这个类的直接父类
        System.out.println(personClass.getGenericSuperclass());
        /**
         * getResourceAsStream 这个方法可以获取到一个输入流，这个输入流会关联到 name 所表示的那个文件上。
         */
        //path 不以’/'开头时默认是从此类所在的包下取资源，以’/'开头则是从 ClassPath 根下获取。其只是通过 path 构造一个绝对路径，最终还是由 ClassLoader 获取资源。
        System.out.println(personClass.getResourceAsStream("/log4j.properties"));
        //默认则是从 ClassPath 根下获取，path 不能以’/'开头，最终是由 ClassLoader 获取资源。
        System.out.println(personClass.getResourceAsStream("/log4j.properties"));

        //判断当前的 Class 对象表示是否是数组
        System.out.println(personClass.isArray());
        System.out.println(new String[3].getClass().isArray());

        //判断当前的 Class 对象表示是否是枚举类
        System.out.println(personClass.isEnum());
        System.out.println(Class.forName("cn.java.reflect.City").isEnum());

        //判断当前的 Class 对象表示是否是接口
        System.out.println(personClass.isInterface());
        System.out.println(Class.forName("cn.java.reflect.TestInterface").isInterface());


    }
}
```

.1.2. 动态代理

在之前的代码调用阶段，我们用 action 调用 service 的方法实现业务即可。

由于之前在 service 中实现的业务可能不能够满足当先客户的要求，需要我们重新修改 service 中的方法，但是 service 的方法不只在我们这个模块使用，在其他模块也在调用，其他模块调用的时候，现有的 service 方法已经能够满足业务需求，所以我们不能只为了我们

的业务而修改 service，导致其他模块授影响。

那怎么办呢？

可以通过动态代理的方式，扩展我们的 service 中的方法实现，使得在原油的方法中增加更多的业务，而不是实际修改 service 中的方法，这种实现技术就叫做动态代理。

动态代理：在不修改原业务的基础上，基于原业务方法，进行重新的扩展，实现新的业务。

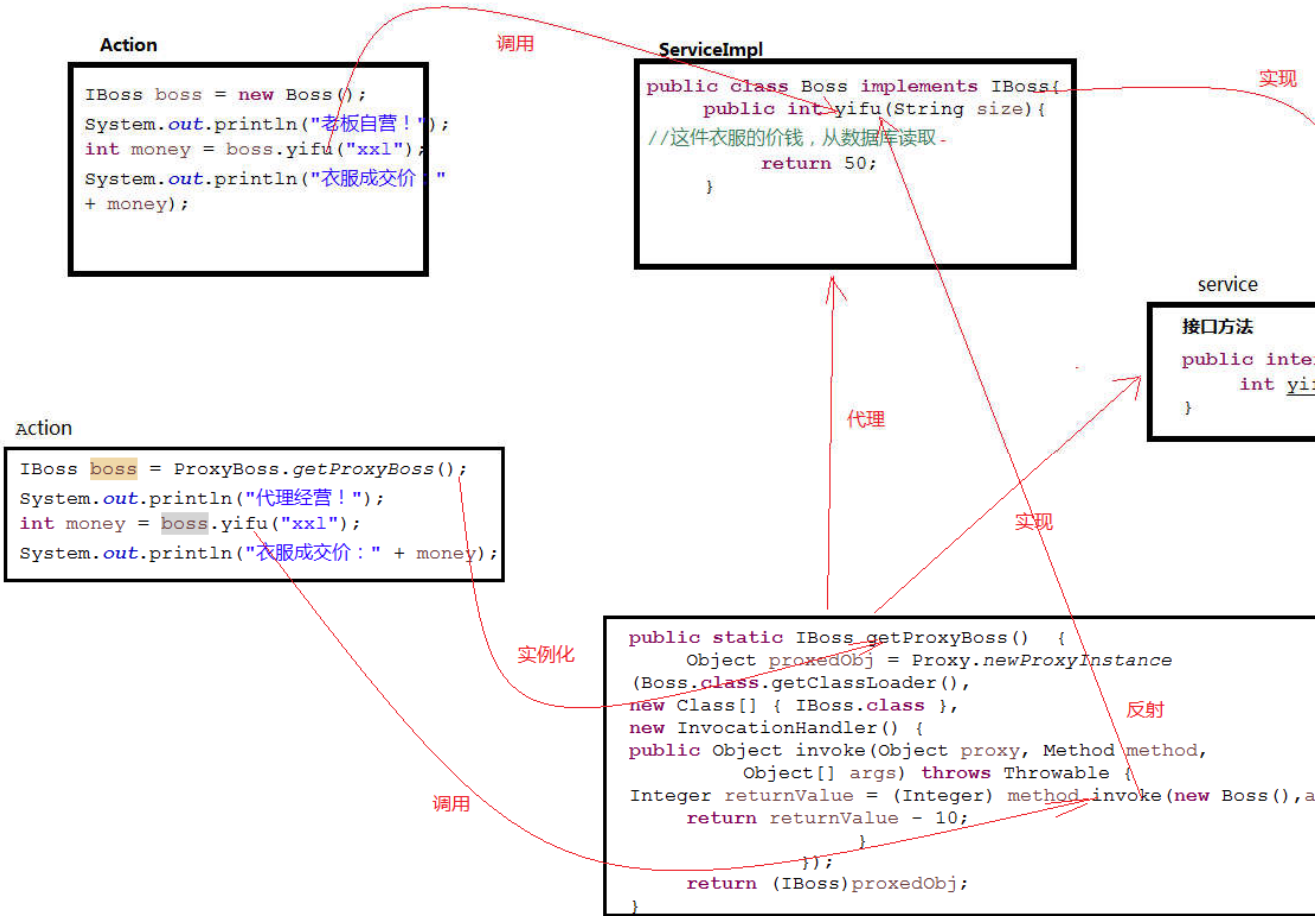
例如下面的例子：

1、 旧业务

买家调用 action，购买衣服，衣服在数据库的标价为 50 元，购买流程就是简单的调用。

2、 新业务

在原先的价格上可以使用优惠券，但是这个功能在以前没有实现过，我们通过代理类，代理了原先的接口方法，在这个方法的基础上，修改了返回值。



代理实现流程：

1、 书写代理类和代理方法，在代理方法中实现代理 Proxy.newProxyInstance

2、 代理中需要的参数分别为：被代理的类的类加载器 soneObjectclass.getClassLoader()，被代理类的所有实现接口 new Class[] { Interface.class }，句柄方法 new InvocationHandler()

3、 在句柄方法中复写 invoke 方法，invoke 方法的输入有 3 个参数 Object proxy（代理类对象）, Method method（被代理类的方法）,Object[] args（被代理类方法的传入参数），在这个方法中，我们可以定制化的开发新的业务。

4、 获取代理类，强转成被代理的接口

5、 最后，我们可以像没被代理一样，调用接口的认可方法，方法被调用后，方法名和参数列表将被传入代理类的 invoke 方法中，进行新业务的逻辑流程。

原业务接口 IBoss

```
public interface IBoss {//接口
    int yifu(String size);
}
```

原业务实现类

```
public class Boss implements IBoss{
    public int yifu(String size){
        System.err.println("天猫小强旗舰店，老板给客户发快递----衣服型号："+size);
        //这件衣服的价钱，从数据库读取
        return 50;
    }
    public void kuzi(){
        System.err.println("天猫小强旗舰店，老板给客户发快递----裤子");
    }
}
```

原业务调用

```
public class SaleAction {
        @Test
    public void saleByBossSelf() throws Exception {
        IBoss boss = new Boss();
        System.out.println("老板自营！ ");
        int money = boss.yifu("xxl");
        System.out.println("衣服成交价： " + money);
    }
}
```

代理类

```
public static IBoss getProxyBoss(final int discountCoupon) throws Exception {
    Object proxedObj = Proxy.newProxyInstance(Boss.class.getClassLoader(),
            new Class[] { IBoss.class }, new InvocationHandler() {
                public Object invoke(Object proxy, Method method,
                        Object[] args) throws Throwable {
                    Integer returnValue = (Integer) method.invoke(new Boss(),
                            args);// 调用原始对象以后返回的值
                    return returnValue - discountCoupon;
                }
            });
    return (IBoss)proxedObj;
}
}
```

新业务调用

```
public class ProxySaleAction {
```

```
        @Test
    public void saleByProxy() throws Exception {
        IBoss boss = ProxyBoss.getProxyBoss(20);// 将代理的方法实例化成接口
        System.out.println("代理经营！");
        int money = boss.yifu("xxl");// 调用接口的方法，实际上调用方式没有变
        System.out.println("衣服成交价：" + money);
    }
}
```

➢ java JVM 技术

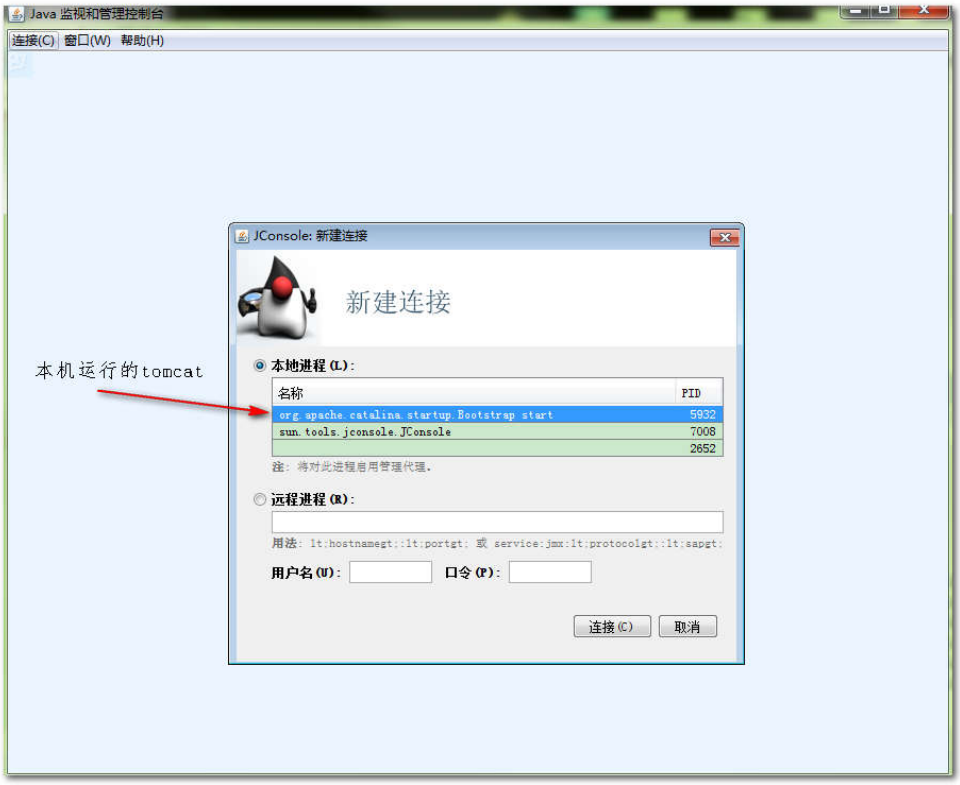
.1. java 监控工具使用

.1.1. jconsole

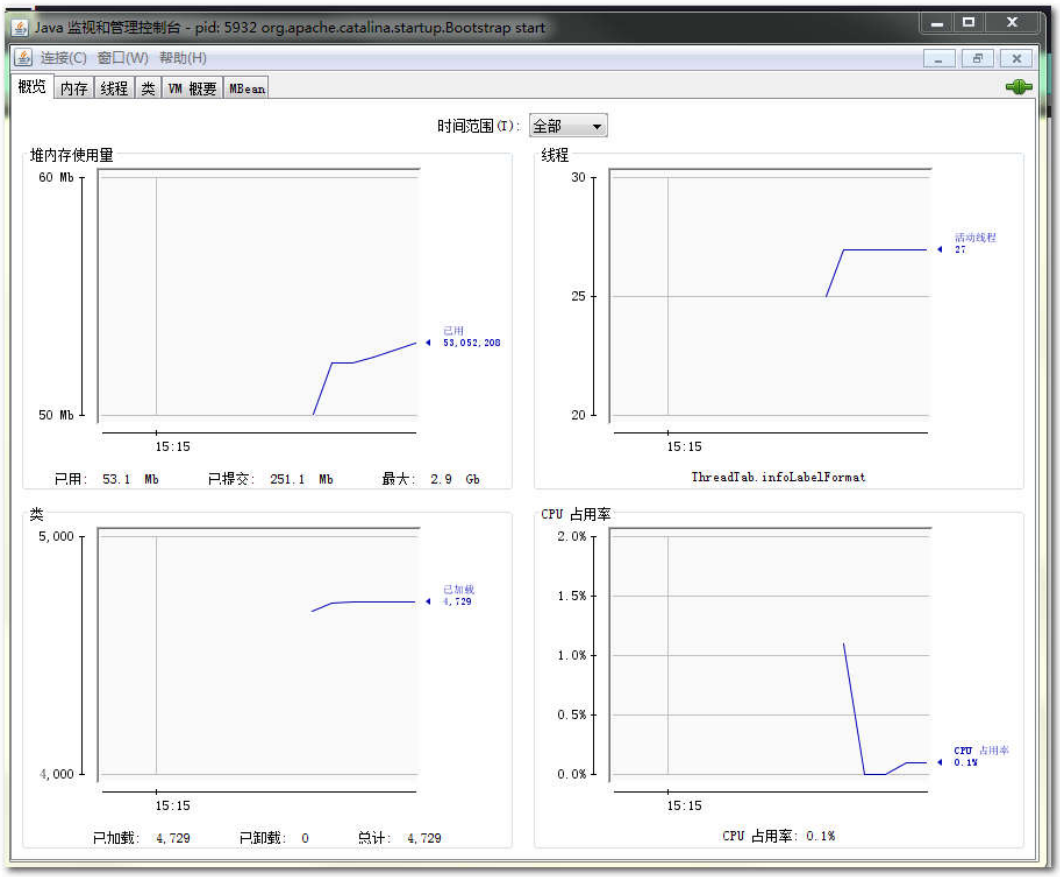
jconsole 是一种集成了上面所有命令功能的可视化工具，可以分析 jvm 的内存使用情况和线程等信息。

启动 jconsole

通过 JDK/bin 目录下的“jconsole.exe”启动 Jconsole 后，将自动搜索出本机运行的所有 JVM 进程，不需要用户使用 jps 来查询了，双击其中一个进程即可开始监控。也可以“远程连接服务器，进行远程虚拟机的监控。”


Java 监视和管理控制台
连接(C) 窗口(W) 帮助(H)
JConsole: 新建连接
新建连接
本机运行的tomcat
本地进程 (L):
名称
PID
org.apache.catalina.startup.Bootstrap start
5932
sun.tools.jconsole.JConsole
7008
2652
注: 将对此进程启用管理代理。
远程进程 (R):
用法: lt;hostnamegt;:lt;portgt; 或 service:jmx:lt;protocolgt;:lt;sapgt;
用户名(U):
口令(P):
连接(C)
取消

概览页面



概述页面显示的是整个虚拟机主要运行数据的概览。

.1.2. jvisualvm

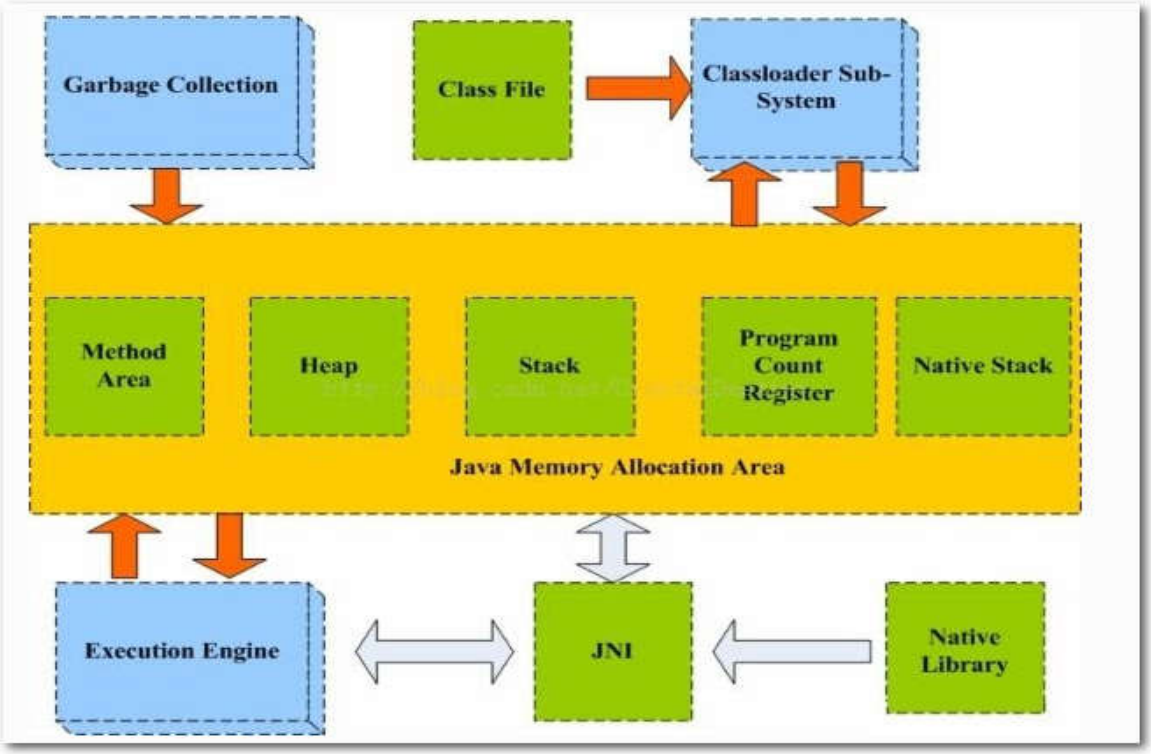
提供了和 jconsole 的功能类似，提供了一大堆的插件。
插件中，Visual GC（可视化 GC）还是比较好用的，可视化 GC 可以看到内存的具体使用情况。

.2. java 内存模型

.2.1. 内存模型图解

Java 虚拟机在执行 Java 程序的过程中，会把它所管理的内存划分为若干个不同的数据区。这些区域有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而存在，有的区域则依赖用户线程的启动和结束而建立和销毁，我们可以将这些区域统称为 Java 运行时数据区域。

如下图是一个内存模型的关系图（详情见图：内存划分.png）：



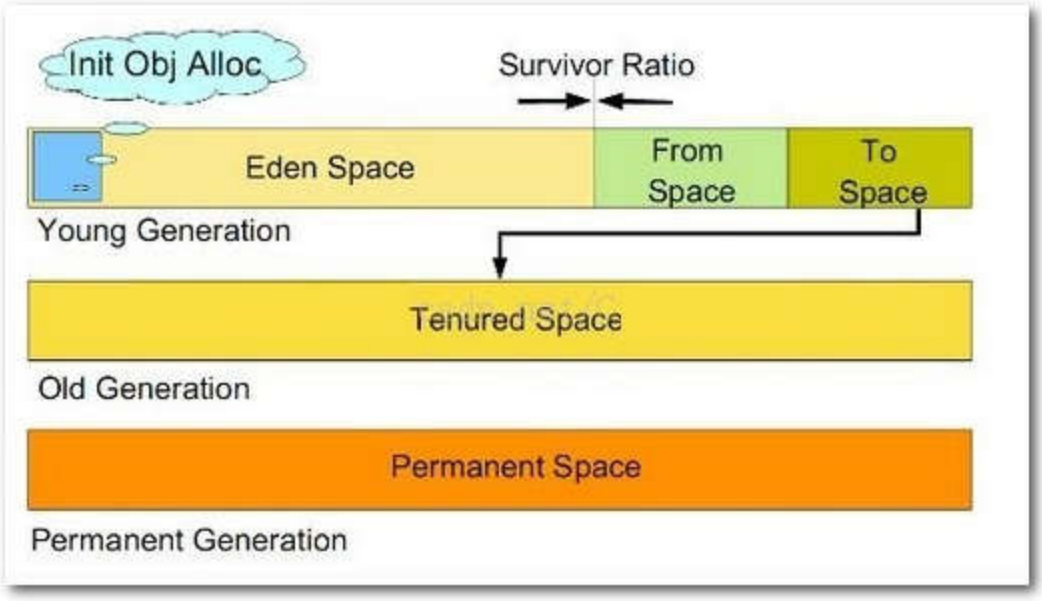
如上图所示，Java 虚拟机运行时数据区域被分为五个区域：堆(Heap)、栈(Stack)、本地方法栈(Native Stack)、方法区(Method Area)、程序计数器(Program Count Register)。

.2.2. 堆（Heap）

对于大多数应用来说，Java Heap 是 Java 虚拟机管理的内存的最大一块，这块区域随着虚拟机的启动而创建。在实际的运用中，我们创建的对象和数组就是存放在堆里面。如果你听说线程安全的问题，就会很明确的知道 Java Heap 是一块共享的区域，操作共享区域的成员就有了锁和同步。

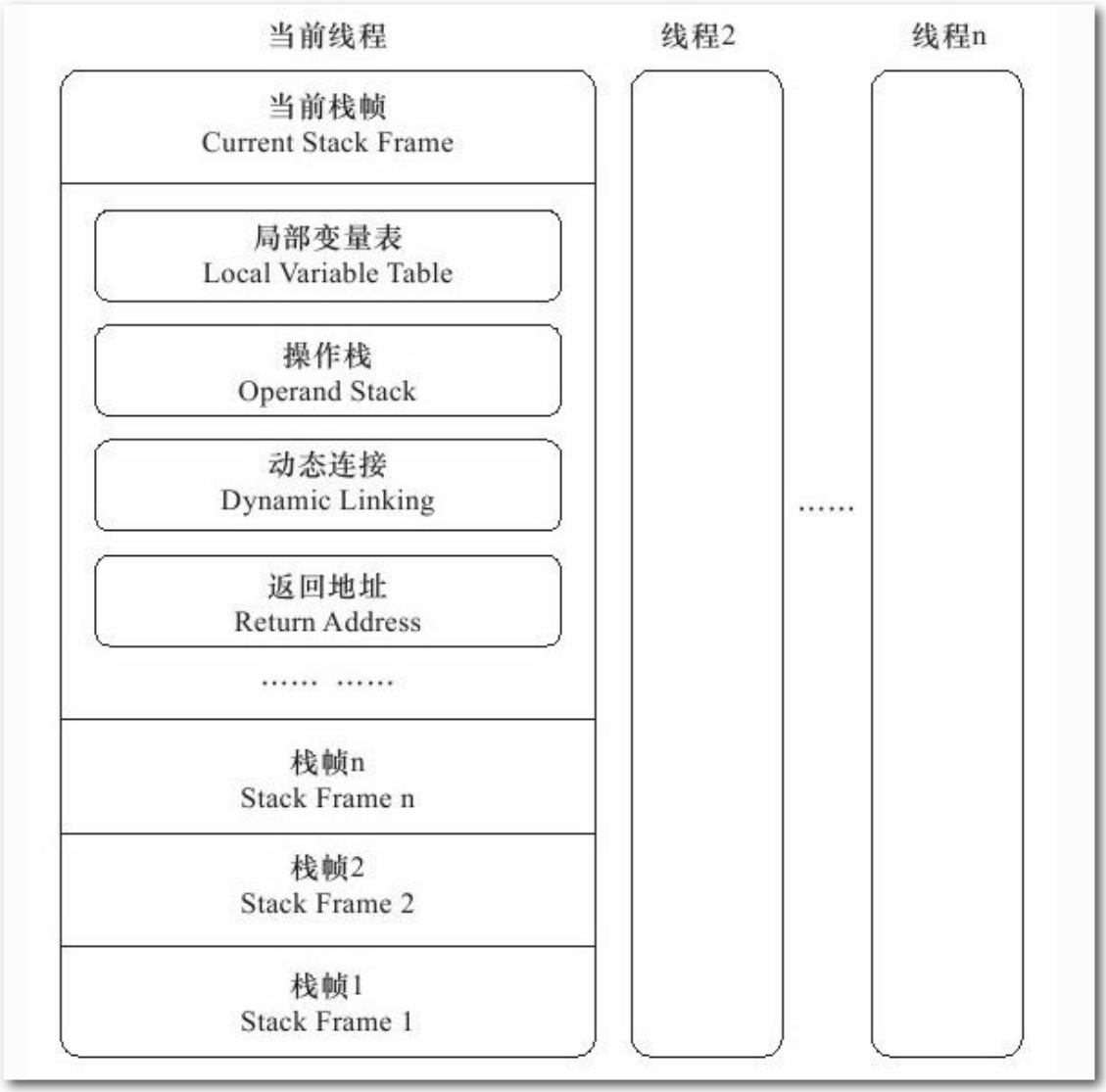
与 Java Heap 相关的还有 Java 的垃圾回收机制（GC）,Java Heap 是垃圾回收器管理的主要区域。程序猿所熟悉的新生代、老生代、永久代的概念就是在堆里面，现在大多数的 GC 基本都采用了分代收集算法。如果再细致一点，Java Heap 还有 Eden 空间，From Survivor 空间,To Survivor 空间等。

Java Heap 可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。



.2.3. 栈（Stack）

相对于 Java Heap 来讲，Java Stack 是线程私有的，她的生命周期与线程相同。Java Stack 描述的是 Java 方法执行时的内存模型，每个方法执行时都会创建一个栈帧（Stack Frame）用语存储局部变量表、操作数栈、动态链接、方法出口等信息。从下图从可以看到，每个线程在执行一个方法时，都意味着有一个栈帧在当前线程对应的栈帧中入栈和出栈。



图中可以看到每一个栈帧中都有局部变量表。局部变量表存放了编译期间的各种基本数据类型，对象引用等信息。

.2.4. 本地方法栈（Native Stack）

本地方法栈（Native Stack）与 Java 虚拟机站（Java Stack）所发挥的作用非常相似，他们之间的区别在于虚拟机栈为虚拟机栈执行 java 方法（也就是字节码）服务，而本地方法栈则为使用到 Native 方法服务。

.2.5. 方法区（Method Area）

方法区（Method Area）与堆（Java Heap）一样，是各个线程共享的内存区域，它用于存储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。虽然 Java 虚拟机规范把方法区描述为堆的一个逻辑部分，但是她却有一个别名叫做非堆（Non-Heap）。分析下 Java 虚拟机规范，之所以把方法区描述为堆的一个逻辑部分，应该觉得她们都是存

储数据的角度出发的。一个存储对象数据（堆），一个存储静态信息(方法区)。

在上文中，我们看到堆中有新生代、老生代、永久代的描述。为什么我们将新生代、老生代、永久代三个概念一起说，那是因为 HotSpot 虚拟机的设计团队选择把 GC 分代收集扩展至方法区，或者说使用永久代来实现方法区而已。这样 HotSpot 的垃圾收集器就能想管理 Java 堆一样管理这部分内存。简单点说就是 HotSpot 虚拟机中内存模型的分代，其中新生代和老生代在堆中，永久代使用方法区实现。根据官方发布的路线图信息，现在也有放弃永久代并逐步采用 Native Memory 来实现方法区的规划，在 JDK1.7 的 HotSpot 中，已经把原本放在永久代的字符串常量池移出。

.2.6. 总结

1、 线程私有的数据区域有：
Java 虚拟机栈（Java Stack）
本地方法栈（Native Stack）
2、 线程共有的数据区域有：
堆（Java Heap）
方法区

.3. JVM 参数列表

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

-Xmx3550m：最大堆内存为 3550M。

-Xms3550m：初始堆内存为 3550m。

此值可以设置与-Xmx 相同，以避免每次垃圾回收完成后 JVM 重新分配内存。

-Xmn2g：设置年轻代大小为 2G。

整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为 64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun 官方推荐配置为整个堆的 3/8。

-Xss128k：设置每个线程的堆栈大小。

JDK5.0 以后每个线程堆栈大小为 1M，在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。

-XX:NewRatio=4:设置年轻代（包括 Eden 和两个 Survivor 区）与年老代的比值（除去持久代）。设置为 4，则年轻代与年老代所占比值为 1：4，年轻代占整个堆栈的 1/5

-XX:SurvivorRatio=4：设置年轻代中 Eden 区与 Survivor 区的大小比值。

设置为 4，则两个 Survivor 区与一个 Eden 区的比值为 2:4，一个 Survivor 区占整个年轻代的 1/6

-XX:MaxPermSize=16m:设置持久代大小为 16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。

如果设置为 0 的话，则年轻代对象不经过 Survivor 区，直 接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在 Survivor 区进行多次复制，这样可以增加对象 再年轻代的存活时间，增加在年轻代即被回收的概论。

收集器设置
-XX:+UseSerialGC:设置串行收集器
-XX:+UseParallelGC:设置并行收集器
-XX:+UseParalledlOldGC:设置并行年老代收集器
-XX:+UseConcMarkSweepGC:设置并发收集器
垃圾回收统计信息
-XX:+PrintGC
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-Xloggc:filename
并行收集器设置
-XX:ParallelGCThreads=n:设置并行收集器收集时使用的 CPU 数。并行收集线程数。
-XX:MaxGCPauseMillis=n:设置并行收集最大暂停时间
-XX:GCTimeRatio=n:设置垃圾回收时间占程序运行时间的百分比。公式为 1/(1+n)
并发收集器设置
-XX:+CMSIncrementalMode:设置为增量模式。适用于单 CPU 情况。
-XX:ParallelGCThreads=n:设置并发收集器年轻代收集方式为并行收集时，使用的 CPU 数。并行收集线程数。

.4. jvm 案例演示

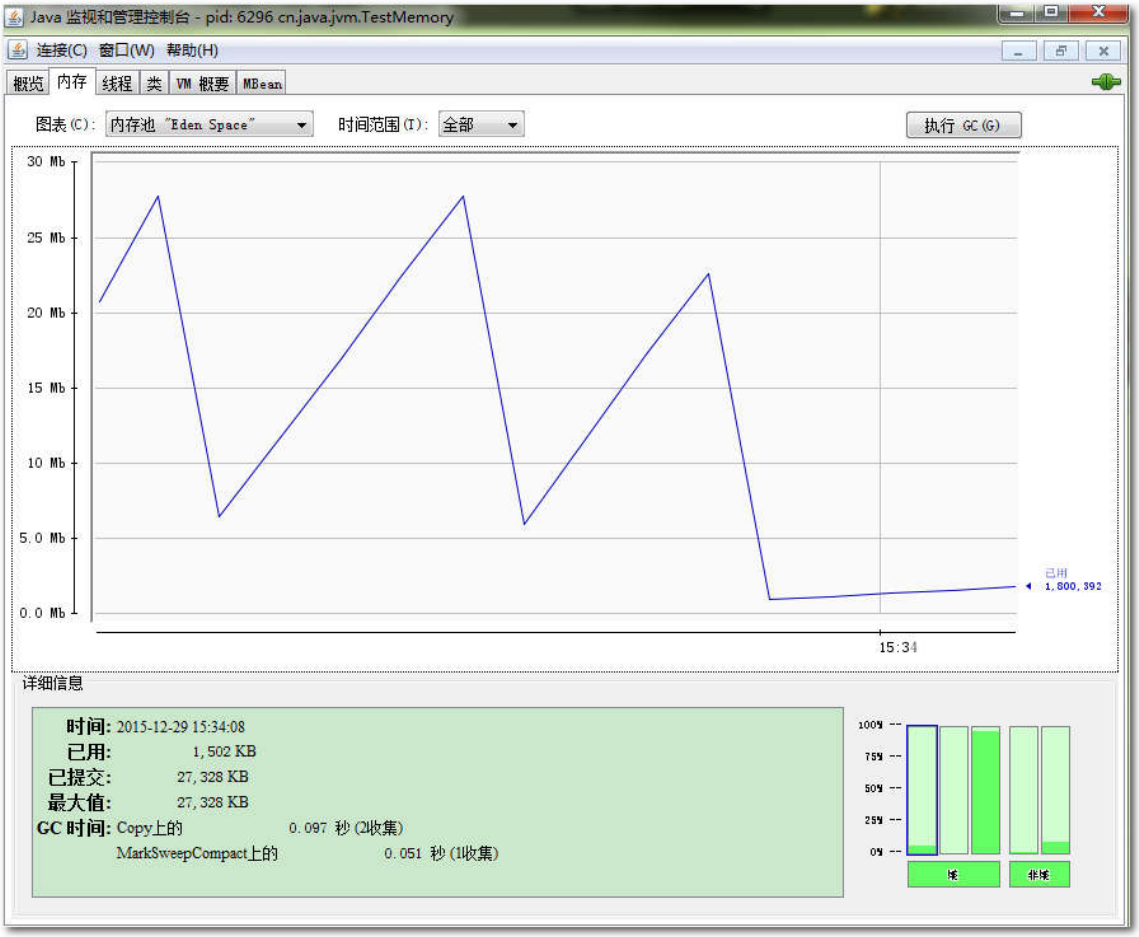
内存：

Jconsole 的内存标签相当于可视化的 jstat 命令，用于监视收集器管理的虚拟机内存（java 堆和永久代）的变化趋势。

我们通过下面的一段代码体验一下它的监视功能。运行时设置的虚拟机参数为：-Xms100m -Xmx100m -XX:+UseSerialGC，这段代码的作用是以 64kb/50 毫秒的速度往 java 堆内存中填充数据。

```
public class TestMemory {
    static class OOMObject {
        public byte[] placeholder = new byte[64 * 1024];
    }

    public static void fillHeap(int num) throws Exception {
        ArrayList<OOMObject> list = new ArrayList<OOMObject>();
        for (int i = 0; i < num; i++) {
            Thread.sleep(50);
            list.add(new OOMObject());
        }
        System.gc();
    }
```

```
    public static void main(String[] args) throws Exception {
        fillHeap(1000);
        Thread.sleep(500000);
    }
}
```



从图中可以看出，运行轨迹成曲线增长，循环 1000 次后，虽然整个新生代 Eden 和 Survivor 区都基本上被清空了，但是老年代仍然保持峰值状态，这说明，填充的数据在 GC 后仍然存活，因为 list 的作用域没有结束。如果把 System.gc();移到 fillHeap(1000);后，就可以全部回收掉。

线程：

jconsole 线程标签相当于可视化了 jstack 命令，遇到线程停顿时，可以使用这个也签进行监控分析。线程长时间停顿的主要原因有：等待外部资源（数据库连接等），死循环、锁等待。下面的代码将演示这几种情况：

```
package cn.java.jvm;

import java.io.BufferedReader;
```

```
import java.io.IOException;
import java.io.InputStreamReader;

public class TestThread {
	/**
	 * 死循环演示
	 *
	 * @param args
	 */
	public static void createBusyThread() {
		Thread thread = new Thread(new Runnable() {
			@Override
			public void run() {
				System.out.println("createBusyThread");
				while (true)
					;
			}
		}, "testBusyThread");
		thread.start();
	}

	/**
	 * 线程锁等待
	 *
	 * @param args
	 */
	public static void createLockThread(final Object lock) {
		Thread thread = new Thread(new Runnable() {
			@Override
			public void run() {
				System.out.println("createLockThread");
				synchronized (lock) {
					try {
						lock.wait();
					} catch (InterruptedException e) {
						e.printStackTrace();
					}
				}

			}
		}, "testLockThread");
		thread.start();
	}
```

```
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        br.readLine();
        createBusyThread();
        br.readLine();
        Object object = new Object();
        createLockThread(object);
    }
}
```

main 线程：追踪到需要键盘录入

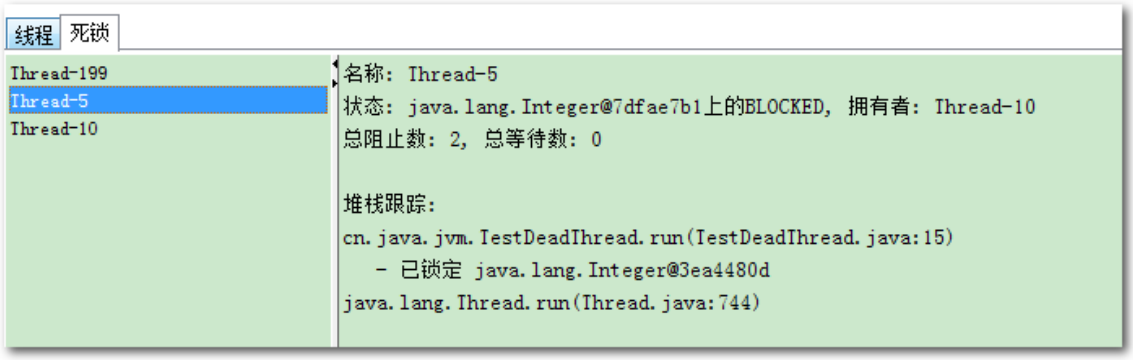
testBusyThread 线程：线程阻塞在 18 行的 while（true），直到线程切换，很耗性能

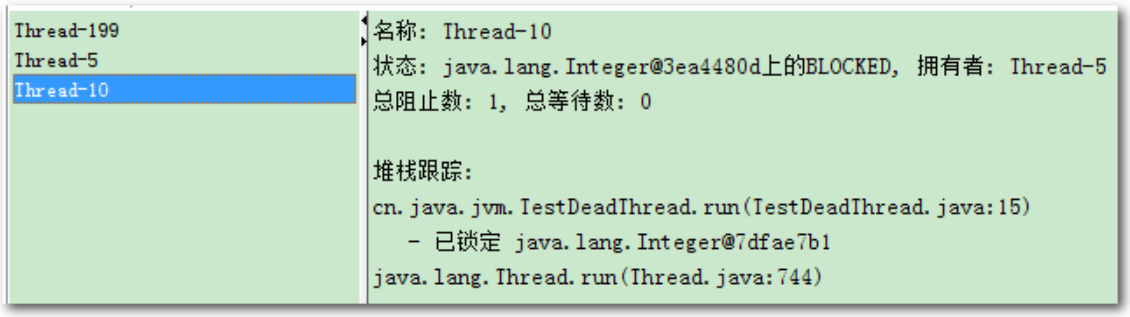
testLockThread 线程：出于 waitting 状态，等待 notify

死锁：

```
package cn.java.jvm;

public class TestDeadThread implements Runnable {
    int a, b;

    public TestDeadThread(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void run() {
        System.out.println("createDeadThread");
        synchronized (Integer.valueOf(a)) {
            synchronized (Integer.valueOf(b)) {
                System.out.println(a + b);
            }
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            new Thread(new TestDeadThread(1, 2)).start();
            new Thread(new TestDeadThread(2, 1)).start();
        }
    }
}
```

点击检查死锁，会出现死锁的详情。





thread-5 的锁被 thread-10 持有，相反亦是，造成死锁。