Java 程序设计进阶-网课

作者

2019

目录

第一章　线程上

课前问题：

1. 程序代码一般都是顺序执行的，如何让程序在运行过程中，实现多个代码段的同时运行？
2. 如何创建多个线程？创建线程的两种方式是什么？
3. 线程为什么要休眠？
4. Thread 类都有哪些方法？

1.1　线程的基本概念

线程：一个线程是一个程序内部的顺序控制流。
线程和进程

- 每个进程都有独立的代码和数据空间（进程上下文），进程切换开销大。
- 线程是轻量的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换的开销小。

多进程：在操作系统中，能同时运行多个任务（程序）。
多进程：在同一应用程序中，有多个顺序流同时执行。

线程的概念模型

1. 虚拟的 CPU，封装在 java.lang.Thread 类中。
2. CPU 所执行的代码，传递给 Thread 类。
3. CPU 所处理的数据，传递给 Thread 类。

线程体

1. Java 的线程是通过 java.lang.Thread 类实现的。
2. 每个线程都是通过某个特定 Thread 对象的方法 run() 来完成其操作，方法 run() 成为线程体。

构造线程的两种方法

1. 定义一个线程类，它继承 Thread 并重写其中的方法 run()；
2. 通过实现接口 Runnable 的类作为线程的目标对象,在初始化一个 Thread 类或者 Thread 子类的线程对象时，把目标对象传递这个线程实例，目标对象提供线程体 run().

   public Thread(ThreadGroup group, Runnable target, String name);

题目

1. Java 中的第一个线程都属于某个线程组；
2. 线程只能在其创建时设置所属的线程组；
3. 线程创建之后，不可以从一个线程组转移到另一个线程组；
4. 新建的线程默认情况下属于其父线程所属的线程组；
5. 线程组成：程序计数器、堆栈、栈指针，进程地址空间中的代码不属于线程。
6. Java 的线程模型：代码可以与其他线程共享，数据可以被多个线程共享，线程模型在 java.lang.Thread 类中被定义。

1.2 通过 Thread 类创建线程

1.2.1 Thread 类

1. Thread 直接继承了 Object 类，并实现了 Runnable 接口，位于 lang.lang。
2. 封装了线程对象需要的属性和方法。

1.2.2 创建线程

1. 从 Thread 类派生一个子类，并创建子类的对象；
2. 子类应该重写 Thread 类的 run() 方法，写入需要在新线程中执行的语句段；
3. 调用 start 方法来启动新线程，自动进入 run 方法。

在 main 中调用新的线程，注：

- main 方法调用 thread.start() 方法启动新线程后并不等待其 run() 方法返回就继续运行，线程的 run 方法在一边独自运行，不影响原来的 main 方法运行。

问题

1. 一个线程是一个 Thread 类的实例，线程从传递给纯种的 Runnable 实例 run() 方法开始执行；
2. 线程操作的数据来自 Runnable 实例，新建的线程调用 start() 方法不能立即进入运行状态，进入就绪状态，必须还要保证当前线程获取到资源以后调用 run() 才能开始执行；
3. 在线程 A 中执行线程 B 的 join() 方法，则线程 A 等待直到 B 执行完成；
4. 线程 A 通过调用 interrupt() 方法来中断其阻塞状态；
5. 若线程 A 调用方法 isAlive() 返回值为 true，则说明 A 正在执行中；
6. currentThread() 获取当前线程对象，而不是引用；
7. 程序的执行完毕与超级线程 (daemon threads) 无关。

1.3 线程的休眠

1.3.1 延长主线程

使用 Thread.sleep(1)，休眠 1ms; 在主线程休眠时，会执行子线程。

对于线程，休眠时会把执行权交给其他线程，如果在 main 线程中调用多个线程，会导致当前线程按照 start 顺序执行。

休眠原因：让其他线程得到执行的机会。

1.3.2 问题

1. Thread.sleep(cnt); 会导致抛出异常；

1.4 Thread 类详解

常用方法见表 1.1

1.5 通过 Runnable 接口创建线程

1.5.1 Runnable 使用及特点

Runnable 接口只有一个 run() 方法，实际上 Thread 也是实现了 Runnable 接口；

表 1.1: Thread 类常用 API 方法

名称	说明
public Thread()	构造一个新的线程对象，默认名为 Thread-n，n 是从 0 开始递增的整数
public Thread(Runnable target)	构造一个新的线程对象，以一个实现 Runnable 接口的类的对象为参数。默认名为 Thread-n，n 是从 0 开始递增的整数。
public Thread(String name)	构造一个新的线程对象，并同时指定线程名
public static Thread currentThread()	返回当前正在运行的线程对象
public static void yield()	使当前线程对象暂停，允许别的线程开始运行
public static void sleep(long millis)	使当前线程暂停运行指定毫秒数，但线程并不失去已获得的锁
public void start()	启动线程，JVM 将调用此线程的 run() 方法，结果是将同时运行两个线程，当前线程和执行 run 方法的线程
public void run()	Thread 的子类应该重写此方法，内容应为该线程应执行的任务
public final void stop()	停止线程运行，释放该线程占用的对象锁
public void interrupt()	中断此线程
public fianl void join()	如果此前启动了线程 A，调用 join 方法将等待线程 A 死亡才能继续执行当前线程
public final void join(long millis)	如果此前启动了线程 A，调用 join 方法将等待指定毫秒数或线程 A 死亡才能继续执行当前线程
public final void setPriority(int new Priority)	设置线程优先级
public final void setDaemon(Boolean on)	设置是否为后台进程，如果当前运行线程均为后台线程，则 JVM 停止运行，此方法必须在 start() 方法调用前使用
public final void checkAccess()	判断当前线程是否有权力修改调用此方法的线程
public void setName(String name)	更改本线程的名称为指定参数
public final boolean isAlive()	测试线程是否处于活动状态，如果线程被启动并且没有死亡则返回 true

- 便于多个线程共享资源

- Java 不支持多继承，如果已经继承了某个基类，便需要实现 Runnable 接口来生成多线程

- 以实现 Runnable 的对象为参数建立新的线程
- start 启动线程

两种方法比较：

- 使用 Runnable 接口

  可以将 CPU、代码和数据分开，形成清晰的模型，还可以从其他类继承

- 直接继承 Thread 类

  编写简单，直接继承，重写 run 方法，不能再从其他类继承

1.5.2 问题

1. java 的线程体由 Thread 类的 run() 方法定义，线程创建时已经确定了提供线程体的对象，java 中每一个线程都有自己的名字

1.6 线程内部的数据共享

用同一个实现了 Runnable 接口的对象作为参数创建多个线程；这些多个线程共享同一个对象中的相同的数据。

使用一个 Runnable 类型对象创建的多个新线程，这多个新线程就共享了这个对象的私有成员。

```
1  //MyThread.java
2  public class MyThread implements Runnable{
3
4      private int sleeptime;
5
6      public MyThread() {
7          sleeptime = (int) (Math.random() * 5000);
8      }
9
10     @Override
11     public void run() {
12         try {
13             System.out.println(Thread.currentThread().getName() + "
                   sleep time: " + sleeptime);
```

```
14              Thread.sleep(sleeptime);
15          } catch (Exception ex) {
16          } finally {
17              System.out.println(Thread.currentThread().getName() + "
                    finished");
18          }
19      }
20  }
21
22  //MyThreadTest.java
23  public class MyThreadTest {
24      public static void main(String[] args) {
25          MyThread mythread = new MyThread();
26          System.out.println("main thread start");
27          new Thread(mythread, "Thread 1").start();
28          new Thread(mythread, "Thread 2").start();
29          new Thread(mythread, "Thread 3").start();
30          System.out.println("main thread end");
31      }
32  }
33
34  //output
35  main thread start
36  main thread end
37  Thread 1 sleep time: 3750
38  Thread 2 sleep time: 3750
39  Thread 3 sleep time: 3750
40  Thread 2 finished
41  Thread 3 finished
42  Thread 1 finished
```

第二章　线程中

课前思考

1. 多线程是如何实现同步的？
2. 多线程中如何避免死锁问题？
3. 线程的生命周期是怎么样的？
4. 多个线程之间优先级如何控制？

学习目标

1. java 中多线程的同步控制方法
2. 掌握线程的生命周期
3. 理解多线程同步的锁机制和线程优先级

2.1　线程同步的思路

2.1.1　多线程的同步控制

- 有时线程之间彼此不独立，需要同步

  线程间的互斥：

  同时运行的几个线程需要共享一些数据

  共享的数据，在某个时刻只允许一个线程对其进行操作

  “生产者/消费者”问题

  假设一个线程负责往数据区写数据，另一个线程从同一数据区读数据，两个线程并行执行

  如果数据区已满，生产者要等消费者取走一些数据后才能再写

  当数据去为空，消费者要等生产者写入一些数据后再取

```
1  //Ticket.java
2  public class Ticket {
3      int number = 0;
4      int size;
5      boolean available = false;
6      public Ticket(int size) {
7          this.size = size;
8      }
9  }
10 //Producer.java
11 public class Producer extends Thread {
12     Ticket t = null;
13     public Producer(Ticket t) {
14         this.t = t;
15     }
16     public void run() {
17         while (t.number < t.size) {
18             System.out.println("Producer puts ticket " + (++t.
                   number));
19             t.available = true;
20         }
21     }
22 }
23 //Consumer.java
24 public class Consumer extends Thread {
25     Ticket t = null;
26     int i = 0;
27     public Consumer(Ticket t) {
28         this.t = t;
29     }
30     public void run() {
31         while (i < t.size) {
32             if (t.available == true && i <= t.number) {
33                 System.out.println("Consumer buys ticket " + (++i))
                       ;
```

```
34              }
35              if (i == t.number) {
36                  t.available = false;
37              }
38          }
39      }
40  }
41  //Machine.java
42  public class Machine {
43      public static void main(String[] args) {
44          Ticket t = new Ticket(10);
45          new Consumer(t).start();
46          new Producer(t).start();
47      }
48  }
49  //output
50  Producer puts ticket 1
51  Producer puts ticket 2
52  Consumer buys ticket 1
53  Producer puts ticket 3
54  Consumer buys ticket 2
55  Producer puts ticket 4
56  Consumer buys ticket 3
57  Producer puts ticket 5
58  Consumer buys ticket 4
59  Producer puts ticket 6
60  Producer puts ticket 7
61  Producer puts ticket 8
62  Consumer buys ticket 5
63  Producer puts ticket 9
64  Producer puts ticket 10
65  Consumer buys ticket 6
66  Consumer buys ticket 7
67  Consumer buys ticket 8
68  Consumer buys ticket 9
69  Consumer buys ticket 10
```

如果将 line 35-line37 改为下面代码，因为当消费者休眠时，CPU 执行生产者 t.available=true，但当消费者休眠完毕，将会设置 t.available=true，因此会导致程序一直运行。

```
1 if (i == t.number) {
2     try {
3         Thread.sleep(1);
4     } catch (InterruptedException e) {
5     }
6     t.available = false;
7 }
```

2.2 线程同步的实现方式-Synchronization

2.2.1 线程同步

- 互斥：许多线程在统一共享数据上操作而不互相干扰，同一时刻只能有一个线程访问该共享数据。因此有些方法或程序段在同一时刻只能被一个线程执行，称之为监视区。
- 协作：多个线程可以有条件地同时操作共享数据。执行监视区代码的线程在条件满足的情况下可以允许其他线程进入监视区。

synchronized

synchronized：线程同步关键字，实现互斥。

- 用于指定需要同步的代码段或方法，也就是监视区。
- 可实现与一个锁的交互
- 功能：首先判断对象锁是否存在，如果存在就获得锁，然后就可以执行紧随其后的代码段；如果对象的锁不存在（已被其他线程拿走），就进入等待状态，直到获得锁。
- 当被 synchronized 限定的代码段执行完，就释放锁。

Java 使用监视器机制

- 每个对象只有一个锁，利用多线程对锁的争夺实现线程间的互斥
- 当线程 A 获得一个对象的锁后，线程 B 必须等待线程 A 完成规定的操作，并释放锁后，才能获得该对象的锁，并执行线程 B 中的操作。

使用锁将生产和消费变为原子操作，解决上节的问题：

```
1  //生产者
2  synchronized (t) {
3      System.out.println("Producer puts ticket " + (++t.number));
4      t.available = true;
5  }
6  //消费者
7  synchronized (t) {
8      if (t.available == true && i <= t.number) {
9          System.out.println("Consumer buys ticket " + (++i));
10     }
11     if (i == t.number) {
12         try {
13             Thread.sleep(1);
14         } catch (InterruptedException e) {
15             e.printStackTrace();
16         }
17         t.available = false;
18     }
19 }
```

当线程执行到 synchronized 时，检查传入的实参对象，并申请得到该对象的锁，如果得不到，那么线程就被放到一个与该对象锁相对应的等待线程池中，直到该对象的锁被返回，池中等待线程才能重新去获得锁，然后执行。

对指定代码段同步控制，还可以定义整个方法，要在方法定义前加上 synchronized 即可。

同步与锁的要点

- 只能同步方法、代码块，不能同步变量
- 每个对象只有一个锁；
- 类可以同时拥有同步和非同步方法，非同步方法可以被多个线程自由访问而不受锁的限制
- 如果两个线程使用相同的实例来调用 synchronized 方法，那么一次只能有一个线程执行方法，另一个需要等待锁。
- 线程休眠时，所持的任何锁都不会释放。

- 线程可以获得多个锁，如在一个对象的同步方法里面调用另一个对象的同步方法，则获取了两个对象的同步锁。

- 同步损害并发性，应缩小同步范围。

- 使用同步代码块时，应该指定在哪个对象上同步。

使用 synchronized 修饰方法：

```
1  //Ticket.java
2  public synchronized void put() {
3      System.out.println("Producer puts ticket " + (++number));
4      available = true;
5  }
6  public synchronized void sell() {
7      if (available == true && i <= number) {
8          System.out.println("Consumer buys ticket " + (++i));
9      }
10     if (i == number) {
11         available = false;
12     }
13 }
14 //生产者
15 public void run() {
16     while (t.number < t.size) {
17         t.put();
18     }
19 }
20 //消费者
21 public void run() {
22     while (t.i < t.size) {
23         t.sell();
24     }
25 }
```

2.3 线程的等待和唤醒

2.3.1 线程的等待-wait() 方法

- 为了更有效地协调不同线程的工作，需要在线程间建立沟通渠道，通过线程间的“对话”来解决线程间的同步问题
- java.lang.Object 的方法

  wait()：如果当前状态不适合本线程执行，正在执行同步代码（synchronized）的某个线程 A 调用该方法（在对象 x 上），该线程暂停执行而进入对象 x 的等待池，并释放已获得的对象 x 的锁。线程 A 要一直等到其他线程在对象 x 上调用 notify 或 notifyAll 方法，才能够在重新获得对象 x 的锁后继续执行（从 wait 语句后继续执行）

2.3.2 线程的唤醒-notify() 和 notifyAll() 方法

- notify() 随机唤醒一个等待的线程，本线程继续执行

  线程被唤醒之后，还要等发出唤醒消息者释放监视者，这期间关键数据仍可能被改变

  被唤醒的线程开始执行时，一定要判断当前状态是否适合自己运行
- notifyAll() 唤醒所有等待的线程，本线程继续执行

```
1  //Ticket.java
2  public synchronized void put() {
3      if (available) {
4          try {
5              wait();
6          } catch (InterruptedException e) {}
7      }
8      System.out.println("Producer puts ticket " + (++number));
9      available = true;
10     notify();
11 }
12 public synchronized void sell() {
13     if (!available) {
14         try {
15             wait();
```

```
16          } catch (InterruptedException e) {}
17      }
18      System.out.println("Consumer buys ticket " + (number));
19      available = false;
20      notify();
21      if (number == size) {
22          number = size + 1;
23      }
24  }
25  //生产者
26  public void run() {
27      while (t.number < t.size) {
28          t.put();
29      }
30  }
31  //消费者
32  public void run() {
33      while (t.number < t.size) {
34          t.sell();
35      }
36  }
```

2.4 后台线程

后台线程

- 也叫守护线程，通常是为了辅助其他线程而运行的线程。
- 不妨碍程序终止
- 一个进程中只要还有一个前程线程在运行，这个线程就不会结束，如果一个进程中所有前台线程结束，那么后台线程就会结束。
- 垃圾回收是一个后台线程
- 启动 start() 之前，使用 setDaemon(true)，使它变为后台线程

```java
public class DaemonTest {
    public static void main(String[] args) {
        Thread t = new DaemonThread();
        t.setDaemon(true);
        t.start();
    }
}
class DaemonThread extends Thread {
    @Override
    public void run() {
        while (true){
            System.out.println("I am running.");
        }
    }
}
```

打印结果：运行很快就结束了，但有语句输出；但不设置后台会一直执行。

2.5 线程的生命周期与死锁

2.5.1 线程的生命周期

- 线程从产生到消亡的过程；
- 一个线程在任何时刻都处于某种线程状态（thread state）。

1. 诞生状态：线程刚被创建
2. 就绪状态：线程的 start 方法已被执行，线程已准备好运行
3. 运行状态：处理机分配给线程，线程正在运行
4. 阻塞状态：

   在线程发出输入输出请求且必须等待其返回

   遇到 synchronized 标记的方法而未获得锁

   为等候一个条件变量，线程调用 wait() 方法
5. 休眠状态：执行 sleep 方法进入休眠
6. 死亡状态：线程完成或退出

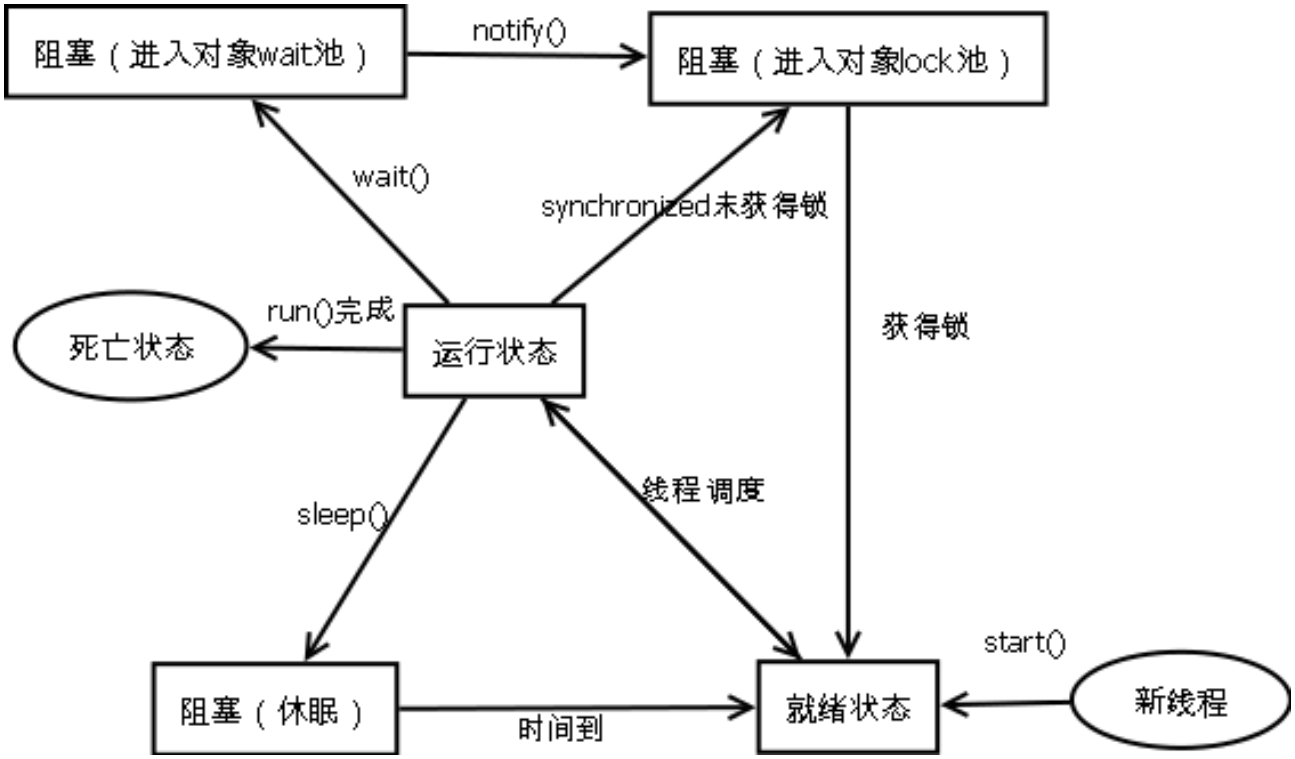

图 2.1: 线程状态生命周期图

2.5.2 死锁

线程在运行过程中，其中某个步骤往往需要满足一些条件才能继续进行下去，如果这个条件不能满足，线程将在此步骤出现阻塞。

如线程 A 等 B，B 等 C，C 又等... 最后回到 A，任何线程不能执行，造成死锁（deadlock）。

2.5.3 控制线程的生命

结束线程的生命：

- 可控制 run 方法中循环条件方式来结束一个线程
- 使用 stop 结束线程生命

  但如果一个线程正在操作共享数据段，操作过程没有完成就用 stop 结束的话，将会导致数据不完整。

2.6 线程的调度

2.6.1 线程的优先级

线程调度

1. 在但 CPU 的系统中，多个线程需要共享 CPU，在任何时间点上实际只能有一个线程在运行；
2. 控制多个线程在同一个 CPU 上以某种顺序运行称为线程调度；
3. Java 虚拟机支持一种非常简单的、确定的调度算法——固定优先级算法，基于线程优先级对其进行调度；
4. 每个 Java 线程都有一个优先级，范围 1-10，默认为 5；
5. 在线程 A 中创建 B，则 B 初始优先级和 A 相同；
6. 如果 A 为后台线程，B 也为后台线程；
7. 可在线程创建后任何时刻，通过 setPriority(int priority) 改变优先级;

基于优先级线程调度

1. 高优先级线程比低优先级先执行；
2. 对相同优先级，Java 处理是随机的；
3. 底层操作系统支持优先级可能小于 10，这会造成一些混乱，只能使用优先级作为作为粗略工具，最后可以通过 yield() 完成。

线程优先级队列
如果某线程正在运行，则只有出现以下情况之一，才会暂停执行

- 一个更高优先级线程处于就绪状态；
- 由于输入输出、调用 sleep、wait、yield 使其阻塞；
- 对于支持时间分片，时间分片执行完。

注：

1. JVM 本身不支持某个线程抢夺另一个正在执行的线程具有同优先级的执行权；
2. yield() 会使正运行线程放弃执行，同优先级线程有机会调度，低优先级仍会被忽略。

第三章　线程下

课前思考

- 线程安全描述的是谁的特性？
- 如何实现线程安全？
- 线程的锁如何优化？

3.1　线程安全与线程兼容与对立

3.1.1　线程安全

线程安全：当多个线程访问同一个对象时，如果不用考虑这些线程在运行时环境的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那这个对象是线程安全的。
Java 的线程安全：

- 不可变
- 绝对线程安全
- 相对线程安全
- 线程兼容和对立

3.1.2　不可变

- final 修饰：public final a = 100;
- java.lang.String: String s = ”string”;
- 枚举类型：public enum ColorRED,BLUE
- java.lang.Number 的子类，如 Long，Double
- BigInteger,BigDecimal（数值类型的高精度实现）

3.1.3 绝对线程安全

- 满足线程安全定义的；
- Java API 标注自己是线程安全的类绝大部分不是绝对线程安全的（java.util.Vector）

3.1.4 相对线程安全

通常意义上的线程安全，需要保证这个对象单独操作是线程安全的，调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就需要在调用时使用同步手段保证调用的正确性。（如：Vector、HashTable）

```
1  public class VectorSafe {
2  private static Vector<Integer> vector = new Vector<Integer>();
3      public static void main(String[] args) {
4          while (true) {
5              for (int i = 0; i < 10; i++) {
6                  vector.add(i);
7              }
8              Thread remove = new Thread(new Runnable() {
9              @Override
10             public void run() {
11                 for (int i = 0; i < vector.size(); i++) {
12                     vector.remove(i);
13                 }
14             }
15             });
16             Thread print = new Thread(new Runnable() {
17             @Override
18             public void run() {
19                 for (int i = 0; i < vector.size(); i++) {
20                     System.out.println(vector.get(i));
21                 }
22             }
23             });
24             remove.start();
25             print.start();
26             while (Thread.activeCount() > 20);
```

```
27          }
28      }
29  }
```

有时会出错，有时不会出错，出错时为数组下标越界错误。

改进方法 synchronized：

```
1  Thread remove = new Thread(new Runnable() {
2      @Override
3      public void run() {
4          synchronized (vector) {
5              for (int i = 0; i < vector.size(); i++) {
6                  vector.remove(i);
7              }
8          }
9      }
10 });
11 Thread print = new Thread(new Runnable() {
12     @Override
13     public void run() {
14         synchronized (vector) {
15             for (int i = 0; i < vector.size(); i++) {
16                 System.out.println(vector.get(i));
17             }
18         }
19     }
20 });
```

3.1.5 线程兼容和线程对立

线程兼容：对象本身不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全使用。

线程对立：无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。

- Java 中线程对立：Thread 类的 suspend() 和 resume() 方法可能导致死锁。

3.2 线程的安全实现-互斥同步

线程安全的实现方式：互斥同步、非阻塞同步、无同步方案。

3.2.1 互斥同步

同步的互斥实现方式：临界区（Critical Section）、互斥量（Mutex）、信号量（Semaphone）。

第一种方式：Java 中使用 Synchronized：编译后，会在同步块前后形成 monitorenter 和 monitorexit 两个字节码。

1. synchronied 同步块对自己是可重入的，不会将自己锁死；
2. 同步块在已进入的线程执行完之前，会阻塞后面线程的进入。

第二种方式：重入锁 ReentrantLock(java.util.concurrent)。

- 相比 synchronized，重入锁可实现：等待可中断、公平锁、锁可以绑定多个条件。
- Synchronized 表现为原生语法层面的互斥锁，而 ReentrantLock 表现为 API 层面的互斥锁。

使用 ReentrantLock：

```
1  public class BufferInterruptibly {
2      private ReentrantLock lock = new ReentrantLock();
3      public void write() {
4          lock.lock();
5          try {
6              long start = System.currentTimeMillis();
7              System.out.println("开始往这个buff写数据");
8              for (;;) {
9                  if (System.currentTimeMillis() - start > Integer.
                       MAX_VALUE) {
10                     break;
11                 }
12             }
13             System.out.println("写数据完毕");
14         } finally {
15             lock.unlock();
16         }
```

```
17      }
18      public void read() throws InterruptedException {
19          lock.lockInterruptibly();
20          try {
21              System.out.println("从buff读数据");
22          }finally {
23              lock.unlock();
24          }
25      }
26  }
```

对于单核、多核，ReentrantLock 效率更高更稳定。

3.3 线程的安全实现-非阻塞同步

阻塞同步：互斥同步存在的问题是在进行线程阻塞和唤醒所带来的性能问题，这种同步称为阻塞同步（Blocking Synchronization）。
非阻塞同步：不同于悲观并发策略，而是使用基于冲突检测的乐观并发策略，就是先进行操作，如果没有其他线程征用共享数据，则操作成功；否则就是产生了冲突，采取不断重试直到成功为止的策略，这种策略不需要把线程挂起，称为非阻塞同步。
使用非阻塞同步条件：

- 使用硬件处理器指令进行不断重试策略；
  - 测试并设置（Test-and-Set）
  - 获取并增加（Fetch-and-Increment）
  - 交换（Swap）
  - 比较并交换（Compare-and-Swap，简称 CAS）
  - 加载链接，条件存储（Load-Linked,Store-conditional，简称 LL,SC）
    Java 实现类 AtomicInteger，AtomicDouble

```
1   class Counter {
2       private volatile int count = 0;
3       public synchronized void increment() {
4           count++;
5       }
```

```
6       public int getCount() {
7           return count;
8       }
9   }
10  class CounterNon {
11      private AtomicInteger count = new AtomicInteger();
12      public void increment() {
13          count.incrementAndGet();
14      }
15      public int getCount() {
16          return count.get();
17      }
18  }
```

3.4 线程的安全实现-无同步方案

3.4.1 无同步方案-可重入代码

也叫纯代码，相对线程安全来说，可以保证线程安全，可以在代码执行过程中中断它，转而去执行另一段代码，而在控制权返回后，原来的程序不会出现任何错误。

3.4.2 无同步方案-线程本地存储

如果一段代码中所需要的数据必须与其他代码共享，那就看看这些代码共享数据的代码是否能保证在同一个线程中执行，如果能保证，就可以把共享数据的可见范围限定在同一线程之内，这样无需同步也能保证线程之间也不会出现数据争用问题。

```
1   public class SequenceNumber {
2       private static ThreadLocal<Integer> seqNum = new ThreadLocal<
            Integer>(){
3           @Override
4           protected Integer initialValue() {
5               return 0;
6           }
7       };
8       public int getNextNum() {
```

```
9           seqNum.set(seqNum.get() + 1);
10          return seqNum.get();
11      }
12
13      public static void main(String[] args) {
14          SequenceNumber sn = new SequenceNumber();
15          TestClient t1 = new TestClient(sn);
16          TestClient t2 = new TestClient(sn);
17          TestClient t3 = new TestClient(sn);
18          t1.start();
19          t2.start();
20          t3.start();
21      }
22      private static class TestClient extends Thread {
23          private SequenceNumber sn;
24          public TestClient(SequenceNumber sn) {
25              this.sn = sn;
26          }
27          public void run() {
28              for (int i = 0; i < 3; i++) {
29                  System.out.println("thread[" + Thread.currentThread
                        ().getName() + "]sn[" + sn.getNextNum() + "]");
30              }
31          }
32      }
33  }
```

3.5 锁优化

锁优化方式：自旋锁、自适应锁、锁消除、锁粗化、偏向锁。

3.5.1 自旋锁

互斥同步存在的问题：挂起线程和恢复线程都需要转入内核态中完成，这些操作给系统的并发性能带来很大压力。

自旋锁：如果物理机器有一个以上的处理器能让两个或以上线程同时并行执行，那就可以让

后面请求锁的那个线程等待一会，但不放弃处理器的执行时间，看看持有锁的线程是否能很快就会释放锁。为了让线程等待，我们只有让线程执行一个忙循环（自旋），这项技术称为自旋锁，Java 默认自旋次数 10 次。

3.5.2 自适应锁

自适应自旋：自适应意味着自旋的时间不再固定，而是由前一次在同一个锁上的自旋时间及锁拥有者的状态来决定。如果在同一个锁对象上，自旋等待刚刚成功获得锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它允许自旋等待相对更长的一段时间。

3.5.3 锁消除

锁消除：JVM 即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。
判断依据：如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步加锁无需进行。

3.5.4 锁粗化

锁粗化：代码中，同步块范围尽量小，只在共享数据的实际作用域中才进行同步，这样是为了使得同步操作的数量尽可能变小。
另一种情况，如果一系列的连续操作都对同一个对象反复加锁，甚至加锁是出现在循环体中，那即使没有现成争用，频繁的进行互斥同步也会导致不必要的性能消耗，此时只需要将同步块范围扩大，即：锁粗化。

3.5.5 偏向锁

偏向锁目的：消除数据无竞争情况下的同步原语，进一步提高程序运行的性能。偏向锁就是在无竞争的情况下把整个同步都消除掉，连 CAS 操作都不做。

3.6 小结

1. 线程安全、线程兼容与线程对立；
2. 线程安全的实现方式；
3. 锁优化。

第四章　网络编程上

课前思考：

1. 如何把互联网上网页抓取下来？
2. 如何与互联网上网络资源通信？
3. 如何在两个 Java 程序之间建立网络连接？
4. 面向连接与非面向连接的通信方式有什么区别？

学习目标：理解计算机网络编程的概念，掌握如何使用 Java 在一台或多台计算机之间进行基于 TCP/IP 协议的网络通讯。

通过理解 TCP/IP 协议的通讯模型，以 JDK 提供的 java.net 包为工具，掌握各种基于 Java 的网络通讯的实现方法。

难点和重点：

1. 基于 URL 的网络编程
2. 基于 TCP 的 C/S 网络编程
3. 基于 UDP 的 C/S 网络编程

4.1　URL 对象

4.1.1　网络基础知识

- IPV4 地址（32 位，4 字节）
- IPV6 地址（128 位，16 字节）
- 主机名（hostname）
- 端口号（port number）
- 服务类型（service）：http、telnet、ftp、smtp

4.1.2 通过 URL 读取 WWW 信息

```
1  public class URLReader {
2      public static void main(String[] args) {
3          URL cs = null;
4          BufferedReader in = null;
5          try {
6              cs = new URL("http://www.sina.com/");
7              in = new BufferedReader(new InputStreamReader(cs.
                   openStream()));
8              String inputLine;
9              while ((inputLine = in.readLine()) != null) {
10             System.out.println(inputLine);
11             }
12             in.close();
13         } catch (IOException e) {
14             e.printStackTrace();
15         }
16     }
17 }
```

4.1.3 URL 类

URL(Uniform Resource Locator) 一致资源定位器的简称，表示 Internet 上某一资源的地址。
URL 组成：Protocol:resourceName

协议名指明获取资源所使用的传输协议，如 http、ftp、gopher、file 等，资源名则应该是资源的完整地址，包括主机名、端口号、文件名或文件内部的一个引用。

4.1.4 构造 URL 对象

- public URL(String spec)

  URL urlBase = new URL("http://gamelan.com/");

- public URL(URL context, String spec)

  URL gamelan = new URL("http://gamelan.com/pages/");

  URL gamelanGames = new URL(gamelan, "Gamelan.game.html");

URL gamelanNetwork = new URL(gamelan, ”Gamelan.net.html”);

- public URL(String protocol, String host, String file);
- public URL(String protocol, String host, int port, String file);

4.1.5 获取 URL 对象属性

- public String getProtocol()
- public String getHost()
- public String getPort()
- public String getFile()
- public String getRef()

4.2 URL Connection 对象

4.2.1 URL Connection

一个 URL Connection 对象代表一个 URL 资源与 Java 程序的通讯连接，可以通过它对这个 URL 资源读或写
与 URL 的区别

- 一个单向，一个双向；
- 可以查看服务器的响应消息的首部；
- 可以设置客户端请求的首部；

使用 URL Connection 通信一般步骤：

1. 构造一个 URL 对象；
2. 调用 URL 对象的 openConnection() 方法获取对应该 URL 的 URL Connection 对象；
3. 配置 URL Connection 对象；
4. 读取首部字段；
5. 获得输入流读取数据；
6. 获得输出流写入数据；
7. 关闭连接。

4.3 Get 请求与 Post 请求

4.3.1 发送 Get 请求

```
1  public static String sendGet(String url, String param){
2      String result = "";
3      BufferedReader in = null;
4      String urlName = url + "?" + param;
5      try {
6          URL realUrl = new URL(urlName);
7          URLConnection con = realUrl.openConnection();
8          con.setRequestProperty("accept", "*/*");
9          con.setRequestProperty("connection", "Keep-Alive");
10         con.connect();
11         in = new BufferedReader(new InputStreamReader(con.
               getInputStream()));
12         String line;
13         while ((line = in.readLine()) != null) {
14             result += line;
15         }
16     } catch (Exception e) {}
17     finally {
18         try{
19             if (in != null) {
20                 in.close();
21             }
22         } catch (Exception ex){}
23     }
24     return result;
25 }
```

4.3.2 发送 POST 请求

POST 请求的参数通过 URL Connection 的输出流写入参数。

```
1  public static String sendPost(String url, String param){
2      PrintWriter out = null;
3      String result = "";
4      BufferedReader in = null;
5      try {
6          URL realUrl = new URL(url);
7          URLConnection con = realUrl.openConnection();
8          con.setRequestProperty("accept", "*/*");
9          con.setRequestProperty("connection", "Keep-Alive");
10         con.setDoOutput(true);
11         out = new PrintWriter(con.getOutputStream());
12         out.print(param);
13         out.flush();
14         in = new BufferedReader(new InputStreamReader(con.
               getInputStream()));
15         String line;
16         while ((line = in.readLine()) != null) {
17             result += line;
18         }
19     } catch (Exception e) {}
20     finally {
21         try{
22             if (in != null) {
23             in.close();
24         }
25         if (out != null) {
26             out.close();
27         }
28         } catch (Exception ex){}
29     }
30     return result;
31 }
```

4.3.3 HttpURLConnection 类

在 URLConnection 的基础上提供了一系列针对 http 请求的内容

- HTTP 状态码
- setRequestMethod(GET/POST)
- getResponseCode(获取 HTTP 的响应)

4.4 Socket 通信原理

4.4.1 TCP 传输协议

TCP(Transport Control Protocol): 面向连接的能够提供可靠的流式数据传输的协议。
如 java 中的 URL、URLConnection、Socket、ServerSocket 等都使用 TCP 协议通讯。

4.4.2 socket 通讯

网络上的两个程序通过一个双向的通讯连接实现数据的交换，这个双向链路的一段称为一个 socket。
socket 通常用来实现客户方和服务方的连接。

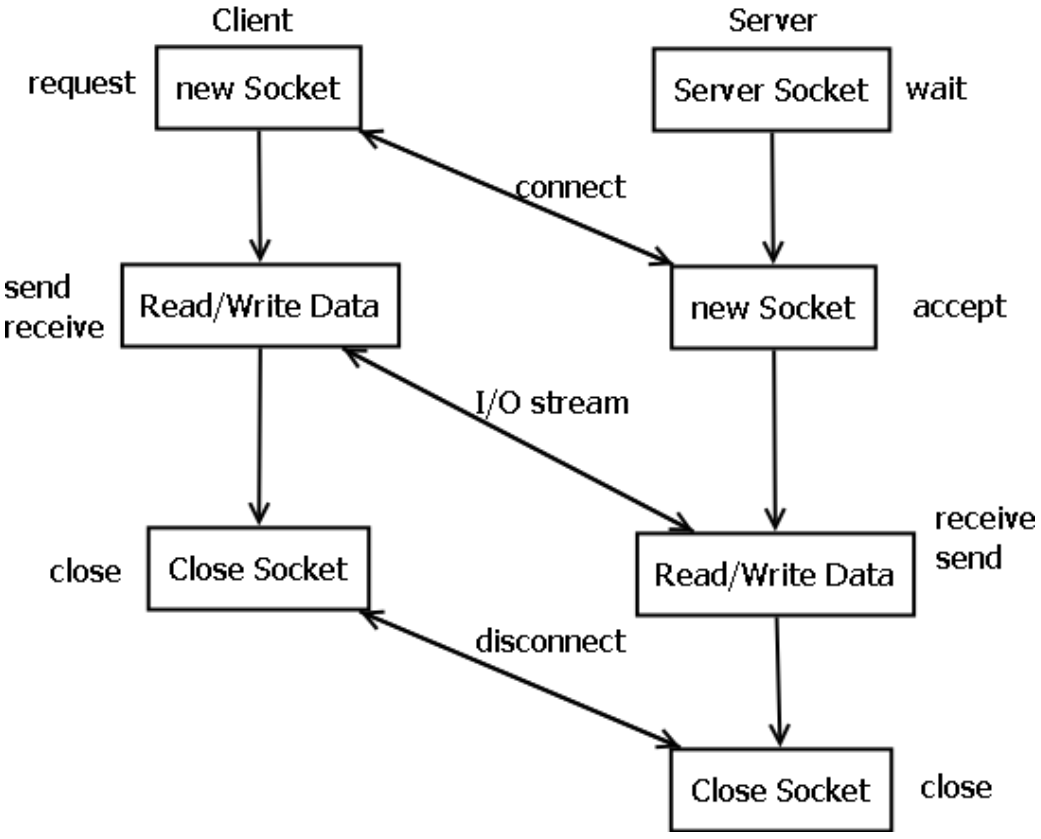

图 4.1: Socket 通信原理

4.5 Socket 通信实现

4.5.1 创建 socket

- Socket()
- Socket(InetAddress address, int port)
- Socket(String host, int port)
- Socket(InetAddress host, int port, InerAddress localAddr,int localPort)
- Socket(String host, int port, InetAddress localAdde, int localPort)

4.5.2 客户端 Socket 的建立

```
1  try {
2      Socket socket = new Socket("127.0.0.1", 2000);
3  } catch (IOException e) {
4      System.out.println("Error: " + e);
5  }
```

4.5.3 服务端 Socket 的建立

```
1  ServerSocket server = null;
2  try {
3      server = new SocketServer(2000);
4  } catch (IOException e) {
5      System.out.println("can not listen to: " + e);
6  }
7  Socket socket = null;
8  try {
9      socket = server.accept();
10 } catch (IOException e) {
11     System.out.println("Error: " + e);
12 }
```

4.5.4 打开输入/输出流

```
1  PrintStream os = new PrintStream(new BufferedOutputStream(socket.
       getOutputStream()));
2  DataInputStream is = new DataInputStream(socket.getInputStream())
       ;
3  PrintWriter out = new PrintWriter(socket.getOutputStream(), true)
       ;
4  BufferedReader in = new BufferedReader(new InputStreamReader(
       socket.getInputStream()));
```

4.5.5 关闭 socket

注意关闭顺序

- os.close();
- is.close();
- socket.close();

客户端和服务器 socket 通信：

```
1  //client
2  public class TalkClient {
3      public static void main(String args[]) {
4          try {
5              Socket socket = new Socket("127.0.0.1",4700);
6              BufferedReader sin = new BufferedReader(new
                   InputStreamReader(System.in));
7              PrintWriter os = new PrintWriter(socket.getOutputStream
                   ());
8              BufferedReader is = new BufferedReader(new
                   InputStreamReader(socket.getInputStream()));
9              String readLine;
10             readLine = sin.readLine();
11             while (!readLine.equals("bye")) {
```

```
12                  os.println(readLine);
13                  os.flush();
14                  System.out.println("Client: " + readLine);
15                  System.out.println("Server: " + is.readLine());
16                  readLine = sin.readLine();
17              }
18              os.close();
19              is.close();
20              sin.close();
21              socket.close();
22          } catch (Exception e) {
23          }
24      }
25  }
```

```
1   //server
2   public class TalkServer {
3       public static void main(String[] args) {
4           try {
5               ServerSocket server = null;
6               server = new ServerSocket(4700);
7               Socket socket = null;
8               socket = server.accept();
9               String line;
10              BufferedReader is = new BufferedReader(new
                    InputStreamReader(socket.getInputStream()));
11              PrintWriter os = new PrintWriter(socket.getOutputStream
                    ());
12              BufferedReader sin = new BufferedReader(new
                    InputStreamReader(System.in));
13              System.out.println("Client: " + is.readLine());
14              line = sin.readLine();
15              while (!line.equals("bye")) {
16                  os.println(line);
17                  os.flush();
```

```
18                  System.out.println("Server: " + line);
19                  System.out.println("Client: " + is.readLine());
20                  line = sin.readLine();
21              }
22              os.close();
23              is.close();
24              sin.close();
25              socket.close();
26              server.close();
27          } catch (Exception e) {
28          }
29      }
30  }
```

第五章　网络编程下

课前思考：

- Java 中 Socket 多客户机制如何实现？
- 数据报是如何通信的？
- 广播通信如何实现？
- 如何实现一个带界面的简单聊天程序？

5.1　Socket 多客户端通信实现

5.1.1　多客户机制

使用多线程

```
1  //服务器线程，客户端与单线程类似
2  public class MultiTalkServer {
3      static int clientnum = 0;
4      public static void main(String[] args) {
5          try {
6              ServerSocket server = null;
7              boolean listening = true;
8              server = new ServerSocket(4700);
9              while(listening){
10                 new ServerThread(server.accept(), clientnum).start
                       ();
11                 clientnum++;
12             }
13             server.close();
```

```
14          } catch (Exception e) {
15          }
16      }
17  }
18  class ServerThread extends Thread {
19      Socket socket = null;
20      int clientnum;
21      public ServerThread(Socket socket,int num) {
22          this.socket = socket;
23          clientnum = num + 1;
24      }
25      public void run() {
26          try {
27          String line;
28          BufferedReader is = new BufferedReader(new
               InputStreamReader(socket.getInputStream()));
29          PrintWriter os = new PrintWriter(socket.getOutputStream());
30          BufferedReader sin = new BufferedReader(new
               InputStreamReader(System.in));
31          System.out.println("Client "+ clientnum + ": " + is.
               readLine());
32          line = sin.readLine();
33          while (!line.equals("bye")) {
34              os.println(line);
35              os.flush();
36              System.out.println("Server: " + line);
37              System.out.println("Client "+ clientnum + ": " + is.
                   readLine());
38              line = sin.readLine();
39          }
40          os.close();
41          is.close();
42          socket.close();
43          } catch (Exception ex) {
44          }
45      }
```

```
46 }
```

5.2 数据报通信

5.2.1 数据报通信

UDP(User Datagram Protocol)：非面向连接的提供的数据包式的数据传输协议。

DatagramPacket,DatagramSocket,MulticastSocket 等类使用 UDP 协议进行网络通讯。

TCP(Transport Control Protocol)：面向连接的能够提供可靠的流式数据传输的协议。

URL,URLConnection,Socket,ServerSocket 等使用 TCP 协议进行网络通讯。

两者区别：

- TCP 有建立时间；
- UDP 传输有大小限制：64K 以内；
- TCP 的应用：Telnet，Ftp；
- UDP 的应用：ping。

Java 使用数据报通信：

- DatagramSocket()
- DatagramSocket(int port)
- DatagramPacket(byte ibuf[],int ilength)
- DatagramPacket(byte ibuf[],int ilength,InetAddress iaddr,int iport)

收数据报：

DatagramPacket packet = new DatagramPacket(buf,256);
socket.receive(packet);

发数据报：

DatagramPacket packet = new DatagramPacket(buf,buf.length,address,port);;
socket.send(packet);

```
1 //客户端
2 public class QuoteClient {
3     public static void main(String[] args) throws Exception {
```

```
4         if (args.length != 1) {
5             System.out.println("Usage: java QuoteClient<hostname>")
                  ;
6             return;
7         }
8         DatagramSocket socket = new DatagramSocket();
9         byte[] buf = new byte[256];
10        InetAddress address = InetAddress.getByName(args[0]);
11        DatagramPacket packet = new DatagramPacket(buf,buf.length,
              address,4445);
12        socket.send(packet);
13
14        packet = new DatagramPacket(buf,buf.length);
15        socket.receive(packet);
16
17        String received = new String(packet.getData());
18        System.out.println("Quote of the Moment: " + received);
19        socket.close();
20    }
21 }
```

```
1  //服务器端
2  public class QuoteServer {
3      public static void main(String[] args) throws Exception {
4          new QuoteServerThread().start();
5      }
6  }
7  class QuoteServerThread extends Thread {
8      protected DatagramSocket socket = null;
9      protected BufferedReader in = null;
10     protected boolean moreQuotes = true;
11     public QuoteServerThread() throws Exception {
12         this("QuoteServerThread");
13     }
14     public QuoteServerThread(String name) throws Exception {
```

```
15          super(name);
16          socket = new DatagramSocket(4445);
17          try {
18              in = new BufferedReader(new FileReader("one-liners.txt"
                    ));
19          } catch (FileNotFoundException e){}
20      }
21      public void run() {
22          while (moreQuotes) {
23              try {
24                  byte[] buf = new byte[256];
25                  DatagramPacket packet = new DatagramPacket(buf,buf.
                        length);
26                  socket.receive(packet);
27                  String dString = null;
28                  if (in == null) {
29                      dString = new Date().toString();
30                  } else {
31                      dString = getNextQuote();
32                  }
33                  buf = dString.getBytes();
34
35                  InetAddress address = packet.getAddress();
36                  int port = packet.getPort();
37                  packet = new DatagramPacket(buf,buf.length,address,
                        port);
38                  socket.send(packet);
39              } catch (IOException e) {
40                  moreQuotes = false;
41              }
42          }
43          socket.close();
44      }
45      protected String getNextQuote() {
46          String returnValue = null;
47          try {
```

```
48              if ((returnValue = in.readLine()) == null) {
49                  in.close();
50                  moreQuotes = false;
51                  returnValue = "No more quote. Goodbye.";
52              }
53          } catch (IOException e) {
54              e.printStackTrace();
55              return returnValue;
56          }
57          return returnValue;
58      }
59  }
```

5.3 使用数据报进行广播通信

DatagramSoctet 只允许数据报发往一个目的地址，MulticastSocket 将数据报以广播方式发往该端口的所有客户，用在客户端，监听服务器广播来的数据。

```
1   //客户端
2   public class MulticastClient {
3       public static void main(String[] args) throws Exception{
4           MulticastSocket socket = new MulticastSocket(4446);
5           InetAddress address = InetAddress.getByName("230.0.0.1");
6           socket.joinGroup(address);
7           DatagramPacket packet;
8
9           for (int i = 0; i < 5; i++) {
10              byte[] buf = new byte[256];
11              packet = new DatagramPacket(buf,buf.length);
12              socket.receive(packet);
13              String received = new String(packet.getData());
14              System.out.println("Quote of the Moment: " + received);
15          }
16          socket.leaveGroup(address);
17          socket.close();
```

```
    }
}
```

```
public class MulticastServer {
public static void main(String[] args) throws Exception {
new MulticastServerThread().start();
}
}
class MulticastServerThread extends QuoteServerThread {
    private long FIVE_SECOND = 5000;
    public MulticastServerThread() throws Exception {
        super("MulticastServerThread");
    }
    public void run() {
        while (moreQuotes) {
            byte[] buf = new byte[256];
            String dString = null;
            if (in == null) {
                dString = new Date().toString();
            } else {
                dString = getNextQuote();
            }
            buf = dString.getBytes();

            try {
                InetAddress group = InetAddress.getByName("
                   230.0.0.1");
                DatagramPacket packet = new DatagramPacket(buf,buf.
                   length,group,4446);
                socket.send(packet);
                sleep((long)(Math.random()*FIVE_SECOND));
            } catch (Exception e) {
                e.printStackTrace();
                moreQuotes= false;
            }
```

```
31          }
32          socket.close();
33      }
34  }
```

5.4 网络聊天程序

```
1  //聊天框
2  public class ChatFrame extends JFrame implements ActionListener {
3
4      JTextField tf;
5      JTextArea ta;
6      JScrollPane sp;
7      JButton send;
8      JPanel p;
9
10     int port;
11     String s = "";
12     String myID;
13     Date date;
14     ServerSocket server;
15     Socket mySocket;
16     BufferedReader is;
17     PrintWriter os;
18     String line;
19
20     public ChatFrame(String ID,String remoteID,String IP,int port,
           boolean isServer){
21         super(ID);
22         myID = ID;
23         this.port = port;
24         ta = new JTextArea();
25         ta.setEditable(false);
26         sp = new JScrollPane(ta);
```

```
27          this.setSize(330,400);
28          this.setResizable(false);
29          try {
30              UIManager.setLookAndFeel(UIManager.
                    getSystemLookAndFeelClassName());
31          } catch (Exception e) {
32              System.out.println("UI error");
33          }
34          this.getContentPane().add(sp,"Center");
35          p = new JPanel();
36          this.getContentPane().add(p,"South");
37          send = new JButton("发送");
38          tf = new JTextField(20);
39          tf.requestFocus();
40          p.add(tf);
41          p.add(send);
42          this.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE
                );
43          this.setVisible(true);
44          send.addActionListener(this);
45          tf.addActionListener(this);
46          if (isServer) {
47              try{
48                  server = null;
49                  try{
50                      server = new ServerSocket(port);
51                  } catch (IOException e) {
52                      e.printStackTrace();
53                  }
54                  mySocket = null;
55                  try {
56                      mySocket = server.accept();
57                  } catch (IOException e) {
58                      e.printStackTrace();
59                  }
```

```
60                  is = new BufferedReader(new InputStreamReader(
                        mySocket.getInputStream()));
61                  os = new PrintWriter(mySocket.getOutputStream());
62              } catch (IOException e) {
63                  e.printStackTrace();
64              }
65          } else {
66              try {
67                  mySocket = new Socket(IP, port);
68                  os = new PrintWriter(mySocket.getOutputStream());
69                  is = new BufferedReader(new InputStreamReader(
                        mySocket.getInputStream()));
70              } catch (Exception e) {
71                  e.printStackTrace();
72              }
73          }
74          while (true) {
75          try {
76              line = is.readLine();
77              date = new Date();
78              SimpleDateFormat formatter = new SimpleDateFormat("yyyy
                    -MM-dd HH:mm:ss");
79              String current = formatter.format(date);
80              s += current + " " + remoteID + "说: \n" + line + "\n";
81              ta.setText(s);
82          } catch (Exception e) {
83              e.printStackTrace();
84          }
85      }
86  }
87
88  @Override
89  public void actionPerformed(ActionEvent e) {
90      date = new Date();
91      SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd
            HH:mm:ss");
```

```
92      String current = formatter.format(date);
93      s += current + " " + myID + "说：\n" + tf.getText() + "\n";
94      ta.setText(s);
95      os.println(tf.getText());
96      os.flush();
97      tf.setText("");
98      tf.requestFocus();
99  }
```

```
1  //服务器
2  public class ChatServerFrame {
3      public static void main(String[] args) {
4          ChatFrame serverFrame = new ChatFrame("Cat","Dog","
               127.0.0.1",2009,true);
5      }
6  }
```

```
1  //客户端
2  public class ChatClientFrame {
3      public static void main(String[] args) {
4          ChatFrame serverFrame = new ChatFrame("Dog","Cat","
               127.0.0.1",2009,false);
5      }
6  }
```

第六章　Java 虚拟机

课前思考：

- Java 语言如何实现跨平台？
- 什么是 Java 虚拟机（JVM）？
- JVM 的内存是如何划分的？
- JVM 的垃圾回收机制都有哪些？

6.1　Java 虚拟机概念

6.1.1　什么是 Java 虚拟机？

- Java 虚拟机是一个想象中的机器，在实际的计算机上通过软件模拟来实现。Java 虚拟机有自己想象的硬件，如处理器、堆栈、寄存器、还有相应的指令系统。

6.1.2　为什么使用 JVM？

- 实现 Java 的跨平台特性
- 把目标代码编译成字节码

分为四层：应用程序层、Java 平台层、操作系统层、硬件层。

6.1.3　Java 虚拟机的生命周期

1. 一个运行中的 Java 虚拟机有着一个清晰的任务：执行 Java 任务。程序开始执行时它才运行，程序结束时它就停止。每个 Java 程序会单独运行一个 Java 虚拟机。

   通过命令行启动 JVM：java classname

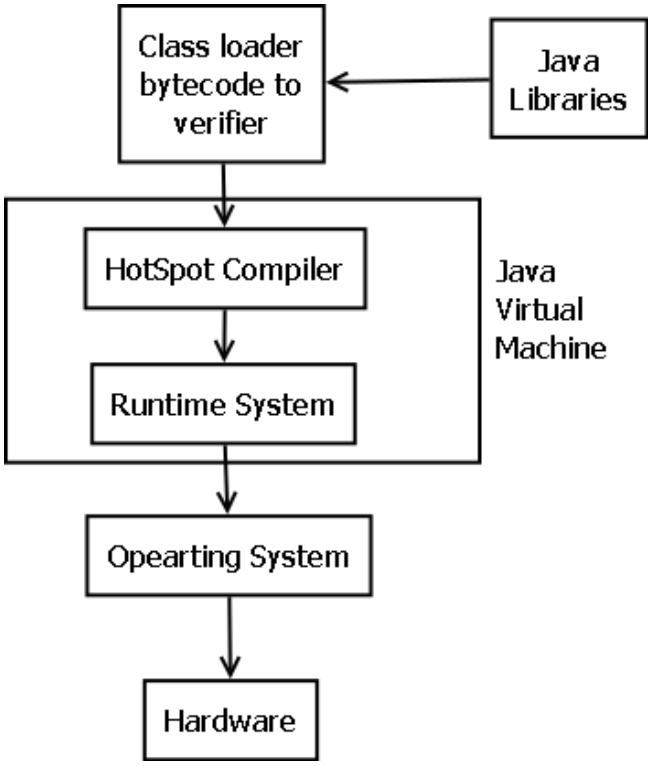

图 6.1: Java 运行环境

2. JVM 总是开始于一个 main() 方法，这个方法必须是 public，返回 void。直接接受一个字符串数组。在程序执行时，必须给 Java 虚拟机指明这个包含有 main() 方法的类名。

   public static void main(String[] args)

3. main() 方法是程序的起点，它被执行的线程初始化为程序的初试线程。程序中其他的线程都由它来启动。Java 中线程分为两种：守护线程（daemon）和普通线程（non-daemon）。

   守护线程是 JVM 自己使用的线程，比如负责垃圾收集的线程，也可以把自己的程序设置为守护线程，包含 main() 方法的初始线程不是守护线程。

4. 只要 JVM 中还有普通线程在执行，JVM 就不会停止，如果有足够的权限，就可以调用 exit() 终止程序。

6.1.4 JVM 的体系结构

在 JVM 的规范中定义了一系列的子系统、内存区域、数据类型和使用指南。这些组件构成了 JVM 的内部结构，它们不仅仅为 JVM 的实现提供了清晰的内部结构，更是严格规定了 JVM 实现的外部行为。

每个 JVM 都有一个类加载子系统（class loader subsystem），负责加载程序中的类型（类 class 和接口 interface），并赋予唯一的名字。每一个 JVM 都有一个执行引擎（execution engine）负责执行被加载类中包含的指令。

6.1.5 JVM 中使用的数据类型

- 所有 JVM 中使用的数据都有确定的数据类型，数据类型和操作都在 JVM 规范中严格定义。Java 中的数据类型分为原始数据类型（primitive types）和引用数据类型（reference type）。

- 在 JVm 中还存在一个 Java 语言中不能使用的原始数据类型–返回值类型（return value）。这种数据类型被用来实现 Java 中的“finally classes”。

- 引用类型可能被创建为：类类型（class type），接口类型（interface type），数组类型（array type）。它们都引用了被动态创建的对象。当引用类型引用 null 时，说明没有引用任何对象。

6.2 Java 虚拟机内存划分

6.2.1 JVM 内存区域

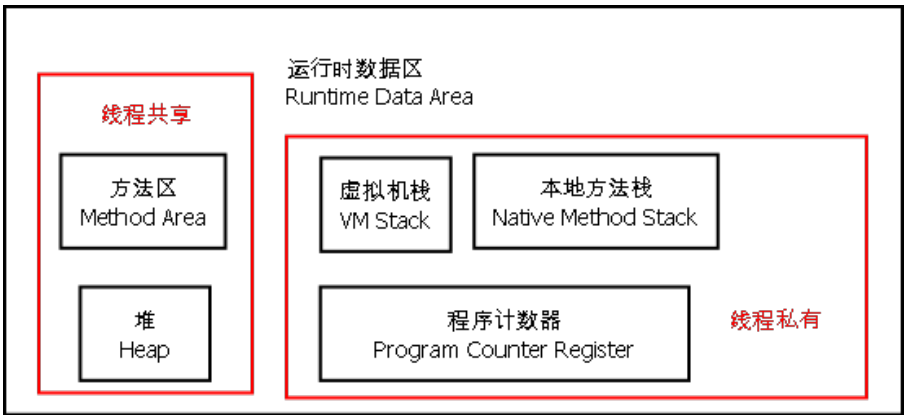

图 6.2: JVM 内存区域

6.2.2 程序计数器

- JVM 将这个计数看作当前执行某条字节码的行数，会根据计数器的值来选去需要执行的操作语句。这个属于线程私有，不可共享，如果共享会导致计数混乱，无法准确的执行当前线程需要执行的语句。

- 该区域不会出现任何 OutOfMemoryError 的情况。

6.2.3 虚拟机栈

- 虚拟机栈就是指经常说到的栈内存。Java 中每一个方法从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入栈到出栈的过程。

- 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），如果扩展时无法申请到足够的内存，就会抛出 OutOfMemoryError 异常。

6.2.4 本地方法栈

- 本地方法栈用来执行本地方法，抛出异常的情况和虚拟机栈一样。而虚拟机栈用来执行 Java 方法。

6.2.5 堆

- 是 JVM 中内存最大、线程共享的一块区域。唯一的目的是存储对象实例。这里也是垃圾收集器主要收集的区域。由于现代垃圾收集器采用的是分带手机算法，所以 Java 堆也分为新生代和老生代。

- 可以通过参数-Xmx(JVM 最大可用内存) 和-Xms(JVM 初始内存) 来调整堆内存，如果扩大至无法继续扩展时，会出现 OutOfMemoryError 的错误。

6.2.6 方法区

- JVM 中内存共享的一片区域，用来存储类信息、常量、静态变量、class 文件。垃圾收集器也会对这部分区域进行回收，比如常量池的清理和类型的卸载。

- 方法区内存不够用的时候，也会抛出 OutOfMemoryError 错误。

6.3 Java 虚拟机类加载机制

6.3.1 虚拟机类加载机制的概念

1. 虚拟机把描述类的数据从 class 文件加载到内存，并对数据进行校验、转换解析和初始化，最初形成可以被虚拟机直接使用的 Java 类型。
2. Java 语言里，类型的加载和连接过程是在程序运行期间完成的。

6.3.2 类的生命周期

- 加载 loading

  通过一个类的全限定名来获取此类的二进制字节码。

  将这个字节码所代表的静态存储结构转化为方法区的运行时数据结构。

  在 Java 堆中生成一个代表这个类的 Class 对象，作为方法区这些数据的访问入口。

- 验证 verfication

  虚拟机规范：验证输入的字节流是否符合 Class 文件的存储格式，否则抛出一个 java.lang.VerifyError 异常。

  文件格式验证：验证字节流是否符合 Classs 文件格式的规范，并且能被当前版本的虚拟机处理。经过这个阶段的验证，字节流进入内存的方法区中进行存储。

  元数据验证：对类的源数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息。

  字节码验证：进行数据流和控制流分析，对类的方法体进行校验分析，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：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），对常量池的各种符号引用的信息进行匹配性的校验。

- 准备 preparation

  准备阶段是正式为类变量分配内存并设置类变量初始值（各数据类型的零值）的阶段，这些内存将在方法区中进行匹配。但是如果类字段的字段属性表中存在 ConstantValue 属性，那在准备阶段变量值就会初始化为 ConstantValue 属性指定的值。

  public static final int value = 122;

- 解析 resoulution

  解析阶段是在虚拟机将常量池内的符号引用替换为直接引用的过程。

  符号引用：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。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。

  直接引用：直接引用是直接指向目标的指针、相对偏移量或者一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

- 初始化 initialization

  <clinit>() 方法：由编译器自动收集类中所有类变量的赋值动作和静态语句块中语句合并并发生，收集的顺序是由语句在源文件中出现的顺序决定的。

该方法与实例构造器 <init>() 不同，不需要显示的调用父类构造器。

<cinit>() 方法对于类或接口来说不是必须的。

执行接口的 <clinit>() 不需要先执行父接口的 <clinit>() 方法。

虚拟机会保证一个类的 <clinit>() 方法在多线程环境中被正确的加锁和同步。

- 使用 using
- 卸载 unloading

6.3.3 类的主动引用

- 遇到 new、getstatic、putstatic、invokestatic 这四个字条码指令时（使用 new 实例化对象的时候、读取或设置一个类的静态字段、调用一个类的静态方法）。
- 使用 java.lang.reflet 包的方法对类进行反射调用的时候。
- 当初始化一个类的时候，如果发现其父类没有进行过初始化，则需要先触发其父类的初始化。
- 当虚拟机启动时，虚拟机会初始化主类（包含 main 方法的那个类）。

6.3.4 类的被动引用

- 通过子类引用父类的静态字段，不会导致子类初始化（对于静态字段，只有直接定义这个字段的类才会被初始化）。
- 通过数组定义类应用类：ClassA[] array = new ClassA[10]。触发了一个名为 LClassA 的类的初始化，它是一个由虚拟机自动生成的、直接继承与 Object 的类，创建动作由字节码指令 newarray 触发。
- 常量会在编译阶段存入调用类的常量池。

6.4 判断对象是否存活算法及对象引用

6.4.1 什么是垃圾回收？

当一个对象没有引用指向它时，这个对象就称为无用的内存（垃圾），就必须进行回收，以便用于后续其他对象的内存分配。

6.4.2 引用计数算法

实现简单，判断效率高，在很大情况下，它都是一个不错算法，但是 Java 语言没有选用引用计数算法管理内存，其中最主要的一个原因是它很难解决对象之间相互循环引用的问题。

ObjA.obj = ObjB;

ObjB.obj = ObjA

6.4.3 可达性分析算法（根搜索算法）

- 在主流的商用程序语言中（Java），都是可达性分析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一张图，从一个节点 GC ROOT 开始，如果一个节点与 GC ROOT 之间没有引用链的存在，该节点视为垃圾回收的对象。

在 Java 中，作为 GC Roots 对象包括：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象；
2. 方法区中的类静态属性引用的对象；
3. 方法区中的常量引用的对象；
4. 本地方法栈中 JNI 的引用的对象。

对象引用-强引用

- 只要引用存在，垃圾回收器永远不会回收

  Object obj = new Object();

- obj 对象对后面 new Object 有一个强引用，只有当 obj 这个引用被释放之后，对象才会被释放掉。

对象引用-软引用

- 非必须引用，内存溢出之前进行回收，可以通过以下代码实现：

```
1 Object obj = new Object();
2 SoftReference<Object> sf = new SoftReference<Obj>(obj);
3 obj = null;
4 sf.get();
```

- 软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

对象引用-弱引用

- 在第二次垃圾回收时，可以通过如下代码实现：

```
1 Object obj = new Object();
2 WeakReference<Object> wf = new WeakReference<Object>(obj);
3 obj = null;
4 wf.get();
5 wf.isEnQueued();
```

- 弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的 isEnqueued 方法返回对象是否被垃圾回收器回收。

虚引用（幽灵/幻影引用）

- 在垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现：

```
1 Object obj = new Object();
2 PhantomReference<Object> pf = new PhantomReference<Object>(obj)
    ;
3 obj = null;
4 pf.get();
5 pf.isEnQueued();
```

- 主要用于检测对象是否已经从内存删除

6.5 分代垃圾回收

6.5.1 分代垃圾回收的提出

- 在 Java 中，没有显示的提供分配内存和删除内存的方法。将引用对象设置为 null 或者调用 System.gc() 来释放内存。
- 在 Java 中，开发人员无法显示删除内存，所以垃圾收集器会发现不需要（垃圾）的对象，然后删除它们，释放内存。

- 基于以上两点，提出分代垃圾收集器。

  1. 绝大数对象在短时间内变得不可达；

  2. 只有少量年老对象引用年轻对象。

6.5.2 年轻代和老年代

年轻代：新创建的对象都存放在这里。因为大多数对象很快变得不可达，所以大多数对象在年轻代中创建，然后消失。当对象从这块内存区域消失时，则成为发生了一次“minor GC”。
老年代：没有变得不可达，存活下来的年轻代对象被复制到这里。这块内存区域一般大于年轻代。因为它更大的规模，GC 发生的次数比年轻代的少。对象从老年代消失时，则称“major GC”（或“full GC”）发生了。

6.5.3 年轻代组成部分

- 年轻代总共有 3 块空间：1 块 Eden 区，2 块 Survivor 区。各个空间的执行顺序如下：

  绝大多数新创建的对象分配在 Eden 区；

  在 Eden 区发生 GC 后，存活的对象移到其中一个 Survivor 区。

  一旦一个 Survivor 已满，存活的对象移动到另外一个 Survivor 区。然后之前那个空间已满 Survivor 区将置为空，没有任何数据。

  经过多次这样步骤依旧存活的对象将被移到老年代。

6.6 典型的垃圾收集算法

6.6.1 Mark-Sweep（标记-清除）算法

最基础的垃圾回收算法，因为它最容易实现，思想也是最简单的。标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记处所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。
缺点：容易产生碎片，碎片太多会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发的一次垃圾手机动作。

6.6.2 Copying(复制) 算法

为了解决 Mark-Sweep 算法的缺陷，Coping 算法就被提出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存使用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。

这种算法虽然实现简单，运行高效且不容易产生内存碎片，但是却对内存空间的使用做出了高昂的代价，因为能够使用的内存缩减到原来的一半。

Copying 算法的效率跟存活的对象的数目多少有很大关系，如果存活对象很多，那么 Copying 算法的效率将会大大降低。

6.6.3 Mark-Compat（标记整理）算法

为了解决 Copying 算法的缺陷，充分利用了内存空间，提出了 Mark-Compact 算法。该算法标记阶段和 Mark-Sweep 一样，但是完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉边界外的内存。

6.6.4 Generational Collection（分代收集）算法

分代收集算法是目前大部分 JVM 的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干不同的区域。一般情况下将堆区划分为老年代（Tenured Feneration）和新生代（Young Generation），老生代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最合适的收集算法。

目前大部分垃圾收集器对于新生代都采取 Copying 算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照 1:1 的比例来划分新生代的空间的，一般来说是将新生代划分为一块较大的 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 空间，当进行回收时，将 Eden 和 Survivor 中还存活的对象复制到另一块 Survivor 空间中，然后清理掉 Eden 和刚才使用过的 Survivor 空间。

对于老年代的特点是每次回收都只回收少量对象，一般使用的是 Mark-Compact 算法。

注意：在堆区之外还有一个代就是永久代（Permanent Generation），它用来存储 class 类、常量、方法描述等。对永生代的回收主要回收两部分内容：废弃常量和无用的类。

6.7 典型的垃圾收集器

6.7.1 Serial/Serial Old

Serial/Serial Old 收集器是最基本最古老的收集器，它是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial 收集器是针对新生代的收集器，采用的是 Copying 算法，Serial Old 收集器是针对老年代的收集器，采用的是 Mark-Compact 算法。它的优点是实现简单高效，但是缺点是会给用户带来停顿。

6.7.2 ParNew

- ParNew 收集器是 Serial 收集器的多线程版本，使用多个线程进行垃圾收集。

6.7.3 Parallel Scavenge

- Parallel Scavenge 收集器是一个新生代的多线程收集器（并行收集器），它在回收期间不需要暂停其他用户线程，其使用的是 Copying 算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

- Parallel Old 是 Parallel Scavenge 收集器的老年代版本（并行收集器），使用多线程和 Mark-Compact 算法。

6.7.4 CMS

CMS（Current Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是 Mark-Sweep 算法。

6.7.5 G1

- G1 收集器是当今收集器技术发展最前沿的成果，它是面向服务端应用的收集器，它能充分利用多 CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。

第七章　深入集合 Collection

课前思考：

1. Java 中有哪些常用集合类？
2. Java 常用集合类如何实现的？
3. 不同集合类在什么场合下使用？
4. 不同集合类其性能如何？

7.1　集合框架与 ArrayList

7.1.1　Java 集合框架

7.1.2　常用集合类

- List

  ArrayList

  LinkedList

- Map

  HashMap

  HashTable

  TreeMap

  LinkedHashMap

- Set

  HashSet

  TreeSet

  LinkedHashSet

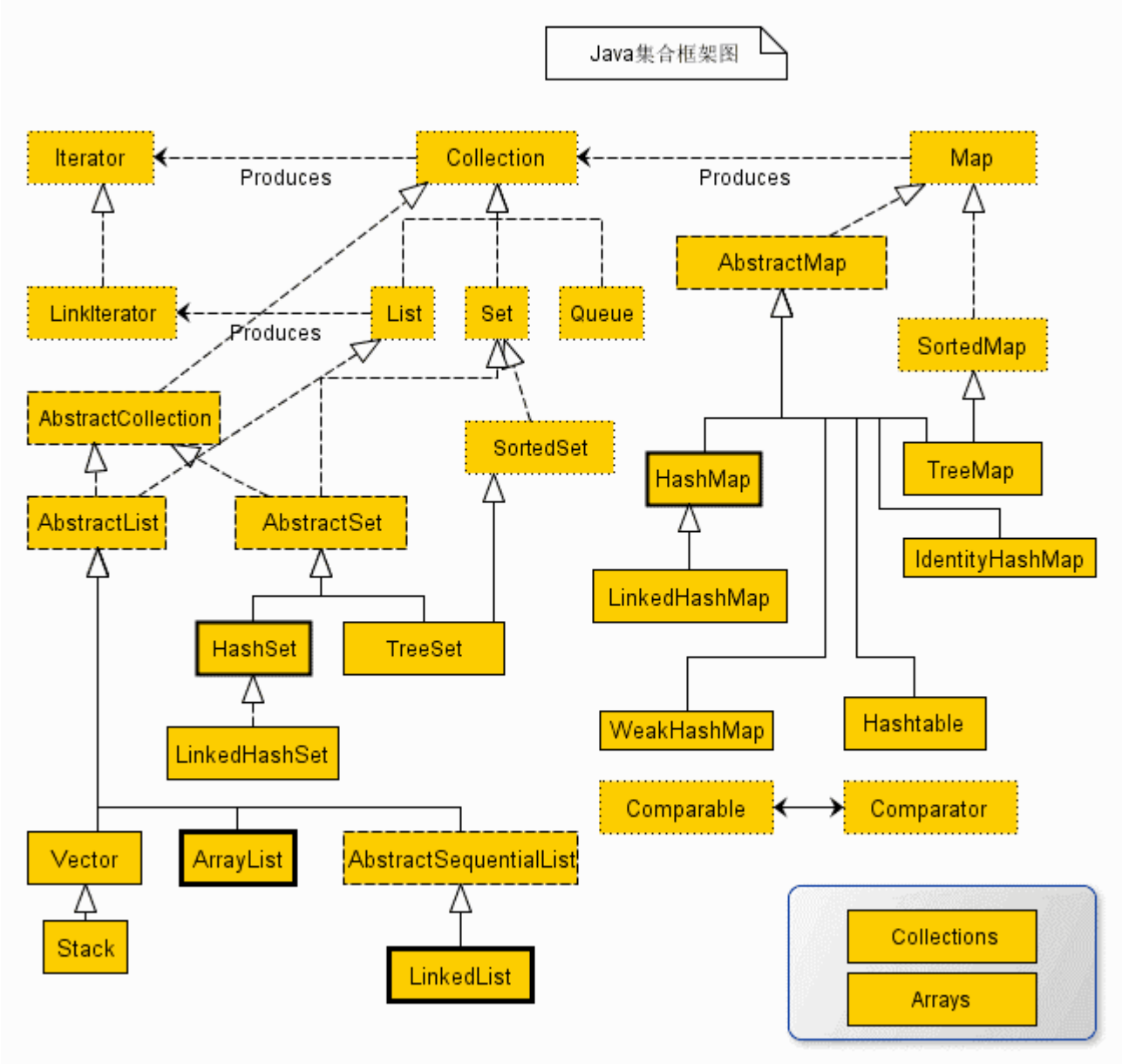

图 7.1: Java 集合框架图

7.1.3 ArrayList

- List 接口的可变数组的实现。实现了所有可选列表操作，并允许 null 在内的所有元素。
- 非线程安全
- 底层使用的数据结构为数组
- 适合查改，弱于增删

7.1.4 ArrayList 实现分析

1. 用指定的元素替代此列表中指定位置上的元素，并返回以前位于该位置上的元素。

```
public E set(int index, E element) {
    rangeCheck(index);

    E oldValue = elementData(index);
    elementData[index] = element;
    return oldValue;
}
```

2. 将指定的元素添加到此列表的尾部。

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}
```

3. 将指定的元素插入此列表中的指定位置。如果当前位置有元素，则向右移动当前位于该位置的元素以及所有后续元素（将其索引加 1），设计数组拷贝，插入速度不及 add(E element) 方法。

```
public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
    size - index);
    elementData[index] = element;
    size++;
}
```

4. 移除此列表中指定位置上的元素。涉及数组拷贝。

```
public E remove(int index) {
    rangeCheck(index);

    modCount++;
    E oldValue = elementData(index);

    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
           numMoved);
    elementData[--size] = null; // clear to let GC do its work

    return oldValue;
}
```

5. 数组扩容，按照 1.5 倍方式扩容。涉及数组拷贝、速度慢。

```
public void ensureCapacity(int minCapacity) {
    //private static final Object[]
       DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};
    //private static final int DEFAULT_CAPACITY = 10;
    int minExpand = (elementData !=
       DEFAULTCAPACITY_EMPTY_ELEMENTDATA)
        // any size if not default element table
        ? 0
        // larger than default for default empty table. It's
           already
        // supposed to be at default size.
        : DEFAULT_CAPACITY;

    if (minCapacity > minExpand) {
        ensureExplicitCapacity(minCapacity);
    }
}

private void ensureExplicitCapacity(int minCapacity) {
```

```
17      modCount++;
18
19      // overflow-conscious code
20      if (minCapacity - elementData.length > 0)
21          grow(minCapacity);
22  }
23
24  private void grow(int minCapacity) {
25  // overflow-conscious code
26  // private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
27      int oldCapacity = elementData.length;
28      int newCapacity = oldCapacity + (oldCapacity >> 1);
29      if (newCapacity - minCapacity < 0)
30          newCapacity = minCapacity;
31      if (newCapacity - MAX_ARRAY_SIZE > 0)
32          newCapacity = hugeCapacity(minCapacity);
33      // minCapacity is usually close to size, so this is a win:
34      elementData = Arrays.copyOf(elementData, newCapacity);
35  }
36
37  private static int hugeCapacity(int minCapacity) {
38      if (minCapacity < 0) // overflow
39          throw new OutOfMemoryError();
40      return (minCapacity > MAX_ARRAY_SIZE) ?
41          Integer.MAX_VALUE :
42          MAX_ARRAY_SIZE;
43  }
```

7.2 LinkedList

- List 接口的链接列表实现，实现所有的列表操作，并且允许所有元素（包括 null）
- 实现了 Deque 接口，为 add、poll 提供先进先出队列操作以及其他堆栈和双端队列操作。
- 非线程安全
- 适合增删，弱于查改。

1. 基于节点 Node 实现：（还有基于 Entry）

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;

    Node(Node<E> prev, E element, Node<E> next) {
        this.item = element;
        this.next = next;
        this.prev = prev;
    }
}
```

2. 根据序号获取 Node 对象

```
public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}

private void checkElementIndex(int index) {
    if (!isElementIndex(index))
    throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
}

private boolean isElementIndex(int index) {
    return index >= 0 && index < size;
}

Node<E> node(int index) {
    // assert isElementIndex(index);

    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
```

```
21              x = x.next;
22          return x;
23      } else {
24          Node<E> x = last;
25          for (int i = size - 1; i > index; i--)
26              x = x.prev;
27          return x;
28      }
29  }
```

3. 在某一个节点前添加元素

```
1   void linkBefore(E e, Node<E> succ) {
2       // assert succ != null;
3       final Node<E> pred = succ.prev;
4       final Node<E> newNode = new Node<>(pred, e, succ);
5       succ.prev = newNode;
6       if (pred == null)
7           first = newNode;
8       else
9           pred.next = newNode;
10      size++;
11      modCount++;
12  }
```

4. 指定位置添加元素，需要先找到 index 的元素，然后添加。

```
1   public void add(int index, E element) {
2       checkPositionIndex(index);
3
4       if (index == size)
5           linkLast(element);
6       else
7           linkBefore(element, node(index));
8   }
```

5. 队首队尾添加元素

```
//队首插入元素
public void addFirst(E e) {
    linkFirst(e);
}
private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}

public void addLast(E e) {
    linkLast(e);
}
//队尾添加元素
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}
```

6. 删除元素

```
public boolean remove(Object o) {
```

```
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null) {
                unlink(x);
                return true;
            }
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item)) {
                unlink(x);
                return true;
            }
        }
    }
    return false;
}
```

实际操作：

```
E unlink(Node<E> x) {
    // assert x != null;
    final E element = x.item;
    final Node<E> next = x.next;
    final Node<E> prev = x.prev;

    if (prev == null) {
        first = next;
    } else {
        prev.next = next;
        x.prev = null;
    }

    if (next == null) {
        last = prev;
    } else {
```

```
17          next.prev = prev;
18          x.next = null;
19      }
20
21      x.item = null;
22      size--;
23      modCount++;
24      return element;
25  }
```

其他删除操作：

```
1   public E removeFirst() {
2       final Node<E> f = first;
3       if (f == null)
4           throw new NoSuchElementException();
5       return unlinkFirst(f);
6   }
7
8   public E removeLast() {
9       final Node<E> l = last;
10      if (l == null)
11          throw new NoSuchElementException();
12      return unlinkLast(l);
13  }
14
15  public E remove(int index) {
16      checkElementIndex(index);
17      return unlink(node(index));
18  }
```

7.2.1 List 的适用范围

- ArrayList 适用于对于数据查询修改大于数据增删的场合；
- LinkedList 适用于对于数据增删大于数据查询的场合。

7.3 HashMap 和 HashTable

7.3.1 HashMap

- 基于哈希表的 Map 接口实现，提供所有可选的映射操作，并允许使用 null 值和 null 键。
- 非线程安全
- 不保证映射的顺序，特别是不保证该顺序恒久不变。

7.3.2 HashMap 数据结构

```
transient Node<K,V>[] table;
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
    this.hash = hash;
    this.key = key;
    this.value = value;
    this.next = next;
    //......
}
```

7.3.3 HashMap 实现分析

```
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
```

```
7       if ((tab = table) == null || (n = tab.length) == 0)
8           n = (tab = resize()).length;
9       if ((p = tab[i = (n - 1) & hash]) == null)
10          tab[i] = newNode(hash, key, value, null);
11      else {
12          Node<K,V> e; K k;
13          if (p.hash == hash &&
14                  ((k = p.key) == key || (key != null && key.equals(k
                        ))))
15              e = p;
16          else if (p instanceof TreeNode)
17              e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key,
                    value);
18          else {
19              for (int binCount = 0; ; ++binCount) {
20                  if ((e = p.next) == null) {
21                      p.next = newNode(hash, key, value, null);
22                      if (binCount >= TREEIFY_THRESHOLD - 1) // -1
                            for 1st
23                          treeifyBin(tab, hash);
24                      break;
25                  }
26                  if (e.hash == hash &&
27                      ((k = e.key) == key || (key != null && key.
                            equals(k))))
28                      break;
29                  p = e;
30              }
31          }
32          if (e != null) { // existing mapping for key
33              V oldValue = e.value;
34          if (!onlyIfAbsent || oldValue == null)
35              e.value = value;
36          afterNodeAccess(e);
37          return oldValue;
38          }
```

```
39      }
40      ++modCount;
41      if (++size > threshold)
42          resize();
43      afterNodeInsertion(evict);
44      return null;
45  }
46
47  static final int hash(Object key) {
48      int h;
49      return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
50  }
```

```
1   public V get(Object key) {
2       Node<K,V> e;
3       return (e = getNode(hash(key), key)) == null ? null : e.value;
4   }
5
6   final Node<K,V> getNode(int hash, Object key) {
7       Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
8       if ((tab = table) != null && (n = tab.length) > 0 &&
9           (first = tab[(n - 1) & hash]) != null) {
10          if (first.hash == hash && // always check first node
11              ((k = first.key) == key || (key != null && key.equals(k
                    ))))
12              return first;
13          if ((e = first.next) != null) {
14              if (first instanceof TreeNode)
15                  return ((TreeNode<K,V>)first).getTreeNode(hash, key
                        );
16              do {
17                  if (e.hash == hash &&
18                  ((k = e.key) == key || (key != null && key.equals(k
                        ))))
19                      return e;
```

```
20                } while ((e = e.next) != null);
21            }
22        }
23        return null;
24    }
```

7.3.4 HashTable 的特点

- HashTable 和 HashMap 采用相同的存储机制，二者的实现基本一致；
- 不允许有 null 值的存在；
- HashTable 是线程安全的，内部实现基本都是 synchronized。
- 迭代器具有强一致性。

7.4 TreeMap 与 LinkedHashMap

7.4.1 TreeMap

- Map 接口的树实现；
- 不允许 null 值的存在；
- 非线程安全；
- 键值有序；
- 使用了红黑树，O(log n) 查找时间复杂度。

7.4.2 TreeMap 实现分析

- Entry 是红黑树的节点，包含了红黑树的 6 个基本组成部分：**key**、**value**、**left**、**right**、**parent**、**color**。Entry 节点根据 key 排序，Entry 点包含的内容为 value。
- 红黑树排序时，根据 Entry 中的 key 进行排序。Entry 的 key 比较大小是根据比较器 comparator 来进行判断的。

7.4.3 TreeMap 的优势

1. 空间利用率高

   HashMap 的数组大小必须为 2 的 n 次方；

   TreeMap 中树的没一个节点就代表了一个元素；

2. 性能稳定

   Hash 碰撞会导致 HashMap 查询开销提高；

   HashMap 扩容时会 rehash，开销高；

   TreeMap 的操作均能在 O(log n) 内完成。

7.4.4 LinkedHashMap

- Map 接口的哈希表和链接列表实现，提供所有可选的映射操作，并允许使用 null 值和 null 键
- 非线程安全
- 具有可预知的迭代顺序

7.4.5 LinkedHashMap 实现

```
Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {
    LinkedHashMap.Entry<K,V> p =
        new LinkedHashMap.Entry<K,V>(hash, key, value, e);
    linkNodeLast(p);
    return p;
}
public V get(Object key) {
    Node<K,V> e;
    if ((e = getNode(hash(key), key)) == null)
        return null;
    if (accessOrder)
        afterNodeAccess(e);
    return e.value;
}
```

7.4.6 Map 的适用范围

- HashMap 适用于一般的键值映射需求；
- HashTable 适用于有多线程并发的场合；
- TreeMap 适用于要按照键排序的迭代场合；
- LinkedHashMap 适用于特殊顺序的迭代场合（如 LRU 算法）。

7.5 HashSet

- 实现 Set 接口，由哈希表支持，允许使用 null 元素；
- 非线程安全；
- 不保证 set 的迭代顺序，特别是不保证该顺序恒久不变

```
1  //底层使用HashMap保存HashSet所有元素
2  private transient HashMap<E,Object> map;
3  //定义一个虚拟的Object对象作为HashMap的Value
4  private static final Object PRESENT = new Object();
5  //借助HashMap的add添加，HashMap的add方法可以返回该key之前的value，
       如果为null则说明之前尚未添加，即HashSet可以添加该元素
6  public boolean add(E e) {
7      return map.put(e, PRESENT)==null;
8  }
9  //借助HashMap的方法来查找
10 public boolean contains(Object o) {
11     return map.containsKey(o);
12 }
```

7.5.1 Set 的特点

- HashSet 通过 HashMap 实现，TreeSet 通过 TreeMap 实现，LinkedHashSet 通过 LinkedHashMap 实现，Set 类与 Map 类拥有近似的使用特性。

第八章　反射与代理机制

课前思考：

1. 给定一个类的名字（字符串形式），怎么创建该类的对象？
2. 什么是反射机制？
3. Java 静态代理和动态代理的异同有哪些？
4. Java 中的类是如何进行加载的？

8.1　Java 反射机制 Reflection

8.1.1　Java 类型信息

- 获取 Java 运行时的类型信息有两种方法：

  RTTI(Run-Time Type Identification)

  Java 反射机制

8.1.2　RTTI

为什么需要 RTTI？

- 在运行中识别一个对象的类型；
- 当存储接口和其实现类时，自动转换为接口。
- 大部分代码尽可能少地了解对象的具体类型，而是只与对象家族中的一个通用表示打交道。

8.1.3　Java 反射机制的定义

- Java 反射机制指的是在运行状态中，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，都能够调用它的任一方法和属性；这种动态获取信息以及动态调用对象方法的功能称为 Java 语言的反射机制。

8.1.4 类 Class

Class 类是 Java 一个基础类，每装载一个新类的时候，JVM 就会在 Java 堆中，创建一个 Class 的实例，这个实例就代表这个 Class 类型，通过实例获取类型信息。该类中的一些方法如下：

- Method[] getMethods()
- Field[] getFields()
- Constructor<?>[] getDeclaredContructors()

Object 类中的方法：

- hashCode()
- equals()
- clone()
- toString()
- notify()
- wait()

8.1.5 利用 Class 类创建实例

- 创建 Class 的一个对象，返回一个类的引用

  Class cls = Class.forName("Airplane");//返回一个类型

- 通过类的引用创建实例

  cls.newInstace(); //通过 newInstace 创建实例，一般调用默认构造函数

```
1 class Airplane {
2     public String toString() {
3         return "in airplane";
4     }
5 }
6
7 class CreateInstance {
```

```
8       public static void main(String[] args) throws
           ClassNotFoundException, IllegalAccessException,
           InstantiationException {
9           Class c1 = null;
10          Object ap;
11          c1 = Class.forName("Airplane");
12          System.out.println(c1);
13          ap = c1.newInstance();
14          System.out.println(ap.toString());
15      }
16  }
17  //output
18  class Airplane
19  in airplane
```

8.1.6 Java 反射例子–Method 类的 invoke

```
1   public class ClassA {
2       public void add(Integer p1, Integer p2){
3           System.out.println(p1.intValue() + p2.intValue());
4       }
5       public void StringAdd(String abc) {
6           System.out.println("out" + abc);
7       }
8
9       public static void main(String[] args) {
10          try {
11              Method mth = ClassA.class.getMethod("add",new Class[]{
                   Integer.class,Integer.class});
12              mth.invoke(ClassA.class.newInstance(),new Integer(1),
                   new Integer(2));
13              Method mth1 = ClassA.class.getMethod("StringAdd",new
                   Class[]{String.class});
14              mth1.invoke(ClassA.class.newInstance(),"--test");
15          } catch (Exception ex){}
```

```
16      }
17  }
```

8.2 Java 静态代理

8.2.1 代理模式

- 在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。
- 代理模式作用：为其他对象提供一种代理以控制对这个对象的访问。

8.2.2 代理模式一般涉及到的角色

- 抽象角色：声明真实对象和代理对象的共同接口。
- 代理角色：代理对象角色内部含有对真实对象的引用，从而操作真实对象，同时代理对象那与真是对象相同的接口以便在任何时刻都能够代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。
- 真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。

8.2.3 静态代理例子

```
1  //真实对象和代理对象的共同接口
2  public abstract class Subject {
3      public abstract void request();
4  }
5  //真实角色
6  public class RealSubject extends Subject {
7      @Override
8      public void request() {
9          System.out.println("From Real Subject");
10     }
11 }
12 //客户端
13 public class Client {
```

```
14      public static void main(String[] args) {
15          Subject subject = new RealSubject();
16          subject.request();
17      }
18  }
19  //代理对象
20  public class ProxySubject extends Subject {
21
22      private RealSubject realSubject;
23
24      @Override
25      public void request() {
26          preRequest();
27          if (realSubject == null) {
28              realSubject = new RealSubject();
29          }
30          realSubject.request();
31          postRequest();
32      }
33      private void preRequest(){
34          System.out.println("Pre Request");
35      }
36      private void postRequest(){
37          System.out.println("Post Request");
38      }
39  }
```

8.2.4 静态代理的优缺点

优点

业务类只需要关注业务逻辑本身，保证了业务类的重用性，这是代理模式共有优点。

缺点

代理对象的一个接口只服务于一种类型的对象，如果要代理的方法很多，势必要为每一种方法都进行代理，静态代理在程序规模稍大时就无法胜任了。

如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法，增加了代码维护的复杂度。

8.3 Java 动态代理

- java.lang.reflect.Proxy

  这是 Java 动态代理机制的主类，它提供了一组动态地生成代理类及其对象。

```
1 //用于获取指定代理对象所关联的调用处理器
2 public static InvocationHandler getInvocationHandler(Object
     proxy)
3 //用于获取关联于指定类装载器和一组接口的动态代理类的类对象
4 public static Class<?> getProxyClass(ClassLoader loader,Class
     <?>... interfaces)
5 //用于判断指定类对象是否是一个动态代理类
6 public static boolean isProxyClass(Class<?> cl)
7 //用于为指定类装载器、一组接口及调用处理器生成动态代理类实例
8 public static Object newProxyInstance(ClassLoader loader,Class
     <?>[] interfaces,InvocationHandler h)
```

- java.lang.reflect.InvocationHandler

  这是调用处理器接口，它自定义了一个 invoke 方法，用于集中处理在动态代理对象上的方法调用，通常在该方法中实现对委托类的代理访问。

```
1 public Object invoke(Object proxy, Method method, Object[] args
     )
2 /**该方法负责集中处理动态代理类上的所有方法调用。第一个参数是代
     理类实例，
3  **第二个参数是被调用的方法对象，第三个参数是调用参数。
4  **调用处理器根据这三个参数进行预处理或分派到委托类实例上执行。
5  */
```

8.3.1 Java 动态代理实例

```
1  //代理接口
2  public interface Subject {
3      public void request();
4  }
```

```
1  //真实角色
2  public class RealSubject implements Subject {
3      public RealSubject(){}
4      @Override
5      public void request() {
6          System.out.println("From real subject");
7      }
8  }
```

```
1  //代理角色
2  public class DynamicSubject implements InvocationHandler {
3
4      private Object sub;
5      public DynamicSubject(){}
6      public DynamicSubject(Object obj) {
7          sub = obj;
8      }
9
10     public Object invoke(Object proxy, Method method, Object[] args
           ) throws Throwable {
11         System.out.println("before calling " + method);
12         method.invoke(sub,args);
13         System.out.println("after calling" + method);
14         return null;
15     }
16 }
```

```
1  //客户端调用
2  public class Client {
3      public static void main(String[] args) {
4          RealSubject rs = new RealSubject();
5          InvocationHandler ds = new DynamicSubject();
6          Class cls = rs.getClass();
7          Subject subject = (Subject) Proxy.newProxyInstance(cls.
               getClassLoader(),cls.getInterfaces(),ds);
8          subject.request();
9      }
10 }
11 //output
12 before calling public abstract void Dynamic.Subject.request()
13 From real subject
14 after calling public abstract void Dynamic.Subject.request()
```

8.3.2 动态代理的特点

1. 包：如果所代理的接口都是 public，那么它将被定义在顶层包，如果所代理的接口非 public，那么它将被定义在该接口所在包，这样设计目的: 为了最大程度的保证动态代理类不会因为包管理的问题而无法被成功定义并访问。

2. 类修饰符：该代理类具有 final 和 public 修饰符，意味着它可以被所有的类访问，但是不能被再度继承；

3. 类名：格式是"$ProxyN"，其中 N 是一个逐一递增的数字，代表 Proxy 类第 N 次生成动态代理类，并不是每次调用 Proxy 的静态方法创建动态代理类都会使得 N 值增加，原因是如果对同一组接口试图重复创建动态代理类，它会很聪明返回先前创建好的代理类的类对象，而不会再尝试创建一个全新的代理类，可以节省不必要的代码重复生成，提高了代理类的创建效率。

4. 类继承关系：

   见下图

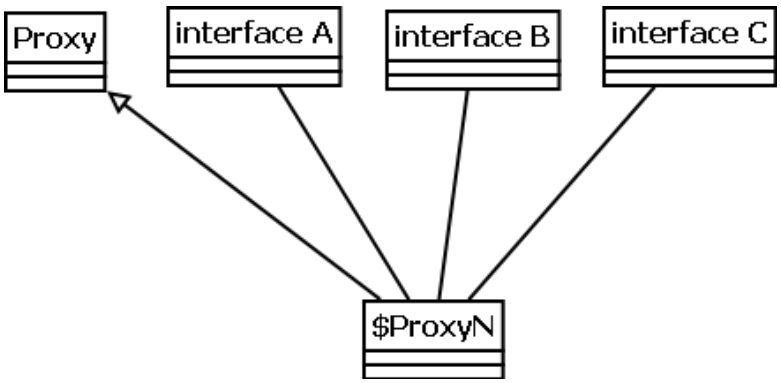

图 8.1: 动态代理类继承关系

8.3.3 动态代理优缺点

优点

动态代理与静态代理相比较，最大优点：接口中声明的所有方法都被转移到调用处理器一个集中的方法中处理（InvocationHandler.invoke）。这样，在接口中方法数量比较多的时候，可以灵活处理，而不需要像静态代理那样每一个方法进行中转。

缺点

始终无法摆脱仅支持 interface 代理的桎梏。

8.4 Java 反射扩展-jvm 加载类原理

8.4.1 JVM 类加载的种类

JVM 自带的默认类加载器

1. 根类加载器：bootstrap，有 C++ 编写，所有 Java 程序无法获得。

2. 扩展类加载器：由 Java 编写。

3. 系统类、应用类加载器：由 Java 编写。

用户自定义的类加载器

java.lang.ClassLoader 的子类，用户可以定制类的加载方式。每个类都包含了加载它的 ClassLoader 的一个引用-getClassLoader()。

如果返回为 null，证明加载它的 ClassLoader 是根加载器 bootstrap。

8.4.2 类的加载方式

- 本地编译好的 class 中直接加载；
- 网络加载：java.net.URLClassLoader 可以加载 url 指定的类；
- 从 jar、zip 等压缩文件加载类，自动解析 jar 文件找到 class 文件去加载；
- 从 Java 源代码文件动态编译成 class 文件。

8.4.3 类加载的步骤

1. 加载；
2. 连接：验证、准备、解析；
3. 类的初始化；

8.4.4 ClassLoader 的加载顺序

加载顺序：

根加载器 -> 扩展类加载器 -> 应用类加载器 -> 用户自定义类加载器

如果到最后一层再加载不了就出现 ClassNotFoundException 异常。

8.4.5 ClassLoader 加载 Class 的过程

1. 检测此 Class 是否加载过，如果有直接到第 8 步，如果没有下一步；
2. 如果 parent classloader 不存在（没有 parent，那 parent 一定是 bootstrap classloader 了），则调到第 4 步；
3. 请求 parent classloader 载入，如果成功到第 8 步，如果不成功第 5 步；
4. 请求 JVM 从 bootstrap classloader 载入，如果成功第 8 步；
5. 寻找 Class 文件（从与此 classloader 相关的类路径中寻找）。如果找不到则到第 7 步；
6. 从文件中载入 Class，到第 8 步；
7. 抛出 ClassNotFoundException;
8. 返回 Class。

8.5 Java 进阶课程总结

8.5.1 Java 线程

- 目的：多程序段执行；
- 安全性、互斥与同步；

8.5.2 Java 的网络编程

- java.net、socket 通信；

8.5.3 集合框架

- 集合框架关键类实现；

8.5.4 JVM

- JVM、垃圾回收；
- Java 反射机制、类加载、类代理。