

## 22 多线程

### ✧ 22.1 进程与线程

**进程**(Process)是资源(CPU、内存等)分配的基本单位，是程序执行时的一个实例。程序运行时系统会创建一个进程，为它分配资源，把该进程放入进程就绪队列，进程调度器选中它时就会为它分配 CPU 时间，程序开始真正运行。

单进程的特定的同一时间段只允许一个程序运行。多进程一个时间段可以运行多个程序，这些程序进行资源的轮流抢占(单核 CPU)，同一个时间点只有一个进程运行。

**线程**(Thread)是程序执行流的最小单位，一个进程可以由多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。线程的启动速度比进程快许多，多线程进行并发处理性能高于多进程。

### ✧ 22.2 继承 Thread 类实现多线程

一个类继承了 `java.lang.Thread` 表示此类是线程的主体类，还需要覆写 `run()` 方法，`run()` 方法属于线程的主方法。多线程要执行的内容都在 `run()` 方法内定义。`run()` 方法不能直接调用，因为牵扯到操作系统资源调度问题，使用 `start()` 方法启动多线程。

```
class MyThread extends Thread {  
    private String name;  
    private int x;  
    public MyThread(String name) {  
        this.name = name;  
    }  
    @Override  
    public void run() {  
        while (this.x < 5) {  
            System.out.println(this.name + ": " + this.x);  
            this.x++;  
        }  
    }  
}  
  
new MyThread("maki").start();  
new MyThread("rin").start();
```

结果:

```
maki: 0  
rin: 0  
maki: 1  
rin: 1  
maki: 2  
rin: 2  
rin: 3
```

```
rin: 4
maki: 3
maki: 4
```

实例化对象调用 `start()` 方法，但是执行的是 `run()` 方法内容，所有线程交替执行，执行顺序不可控，打印结果随机。

为什么要使用 `start()` 方法启动多线程呢？

`start()` 方法源代码：

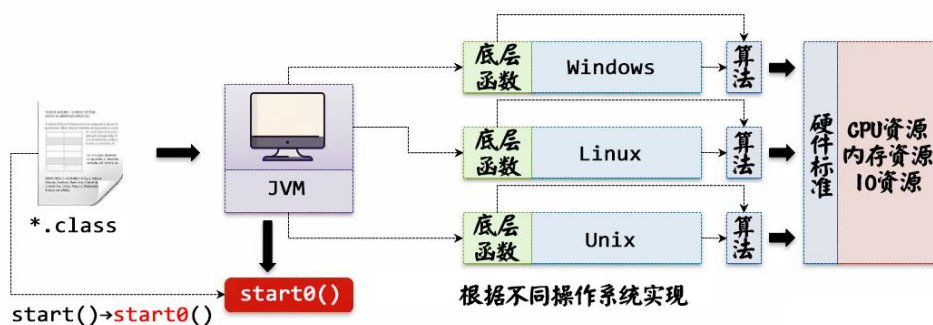
```
public synchronized void start() {
    if (threadStatus != 0) // 线程的状态 0 表示线程未开始
        throw new IllegalStateException();
    group.add(this);
    boolean started = false;
    try {
        start0();
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {}
    }
}

private native void start0(); // 该方法没有方法体,没有实现
```

一个线程只能被启动一次，如果重复启动抛出 `IllegalThreadStateException` 异常。但没有 `throws` 声明或 `try-catch` 处理，说明该异常是 `RuntimeException` 的子类

`start()` 方法中又调用了 `start0()` 方法，此方法使用 `native` 关键字修饰。

Java 支持本地操作系统函数调用，称为 JNI (Java Native Interface) 技术，但是 Java 开发中不推荐这样使用，利用 JNI 可以使用操作系统提供的底层函数。Thread 类的 `start0()` 表示此方法依赖于不同的操作系统实现。



## ✧ 22.3 基于 `Runnable` 接口实现多线程

Java 继承存在单继承的局限，实现 `java.lang.Runnable` 接口也可以实现多线程。

Runnable 接口的定义:

```
@FunctionalInterface // 函数式接口
public interface Runnable {
    public abstract void run();
}
```

将 MyThread 改为实现 Runnable 接口:

```
class MyThread implements Runnable {
    // 与之前一模一样...
}
```

但是此时 MyThread 没有继承 Thread，不能使用 start()方法。

Thread 类有一个构造方法可以接收 Runnable 对象作为参数:

```
public Thread(Runnable target) {
    init(null, target, "Thread-" + nextThreadNum(), 0);
}
```

启动多线程:

```
new Thread(new MyThread("maki")).start();
new Thread(new MyThread("rin")).start();
```

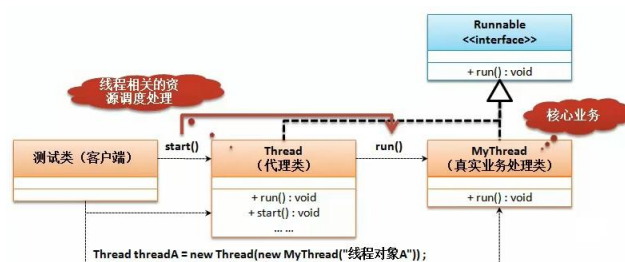
JDK 1.8 开始 Runnable 使用了函数式接口定义，可以使用 Lambda 表达式定义多线程类。

```
for (int i = 0; i < 3; i++) { // 3个线程
    String name = "线程-" + i;
    Runnable run = () -> { // Runnable对象
        for (int j = 0; j < 5; j++) {
            System.out.println(name + ": " + j);
        }
    };
    // 始终使用Thread对象start()方法启动多线程
    new Thread(run).start();
}
```

也可以不定义 run 变量，直接将其右边传入 Thread 的构造方法。

## ✧ 22.4 Thread 类和 Runnable 接口关系

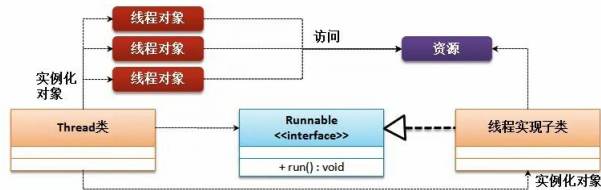
Thread 类是 Runnable 接口的子类



多线程设计使用了代理设计模式的结构,用户自定义的线程主体只是负责核心业务的实现,所有的辅助实现都由 Thread 类处理。

通过 Thread 类的构造方法传递 Runnable 对象时，该对象被 Thread 的 target 属性保存。Thread 启动多线程调用 start()方法，start()调用 run()方法，此 Thread 类的 run()方法又去调用 Runnable 对象的 run()方法。

多线程开发本质的多个线程可以进行同一资源的抢占。Thread 主要描述线程，Runnable 主要描述资源，因为  $n$  个 Thread 对象的 target 属性都指向了同一个 Runnable 对象。



## ✧ 22.5 Callable 实现多线程

Runnable 接口的缺点是当线程执行完毕无法获取返回值。JDK 1.5 提出新的线程实现接口 `java.util.concurrent.Callable`:

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

Callable 对象可以作为 FutureTask 构造方法的参数保存为 callable 属性。

**FutureTask** 是 **RunnableFuture** 接口的子类。

```
public class FutureTask<V> implements RunnableFuture<V> {  
    // ...  
    public FutureTask(Callable<V> callable) {  
        if (callable == null)  
            throw new NullPointerException();  
        this.callable = callable;  
        this.state = NEW; // ensure visibility of callable  
    }  
    // ...  
}
```

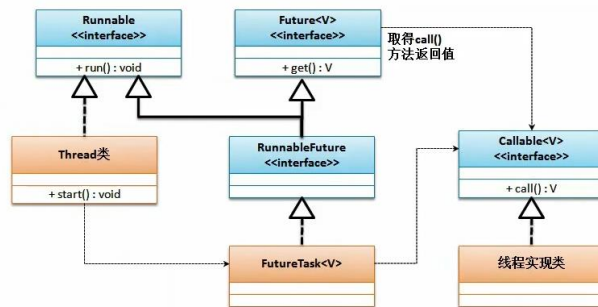
RunnableFuture 接口继承于 Runnable 接口和 Future 接口。

```
public interface Runnable Future<V> extends Runnable, Future<V> {
    void run();
}
```

**FutureTask** 类覆写了 **Future** 接口的 **get()** 方法, 可以获取 callable 属性调用 **call()** 方法的返回值。

`FutureTask` 类也是 `Runnable` 接口的子类，可以作为 `Thread` 构造方法的参数。

关系有点复杂：



示例：用 `Callable` 实现龟兔赛跑

```
class Race implements Callable<Integer> {
    private String name;
    private long time; // 多少毫秒走一步
    private int step; // 步数
    private boolean flag = true; // 设为false结束线程
    public Race(String name, long time) {
        this.name = name;
        this.time = time;
    }
    public void setFlag(boolean flag) {
        this.flag = flag;
    }
    @Override
    public Integer call() throws Exception {
        while (flag) {
            Thread.sleep(this.time);
            this.step++;
            System.out.println(Thread.currentThread().getName() + " " +
this.name + ": " + this.step);
        }
        return this.step;
    }
}

// 客户端代码太多了...
Race tortoise = new Race("乌龟", 2000);
Race rabbit = new Race("兔子", 500);
FutureTask<Integer> task1 = new FutureTask<>(tortoise);
FutureTask<Integer> task2 = new FutureTask<>(rabbit);
new Thread(task1).start();
new Thread(task2).start();
```

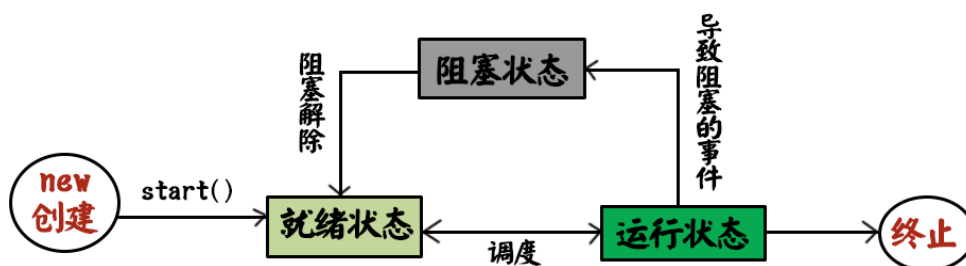
```
Thread.sleep(10000); // 跑10秒
tortoise.setFlag(false);
rabbit.setFlag(false);
System.out.println("10秒后, 乌龟:" + task1.get());
System.out.println("10秒后, 兔子:" + task2.get());
```

结果:

```
10 秒后, 乌龟:5
10 秒后, 兔子:20
```

## ✧ 22.6 多线程运行状态

定义线程主体类, 通过 Thread 类 start()方法启动线程, 但并不是调用 start()方法线程就开始运行。



- 1) 任何一个线程对象需要使用 Thread 类封装, 线程启动使用 start()方法; 但是启动时所有线程进入就绪状态, 并没有执行;
- 2) 等待资源调度, 某个线程调度成功则进入运行状态(run()方法); 但是一个线程不可能一直执行下去, 执行一段时间之后就会让出资源进入阻塞状态, 随后重新回到就绪状态;
- 3) run()方法执行完毕, 线程任务结束, 此时进入停止状态。

20180607

## ✧ 22.7 Thread 类常用方法

### ① 线程的命名与取得

多线程运行状态不确定, 所有线程的名字是个重要的属性。

```
public Thread(Runnable target, String name) // 构造方法可以自定义线程名
public final void setName(String name) // 设置线程名字
public final String getName() // 获取线程名字
```

线程对象的获取不可能只靠 this 完成, 因为线程状态不可控, 但是所有线程一定会执行 run()方法, 则可以考虑获取当前线程。

```
public static Thread currentThread() // 返回当前正在执行线程的引用
```

示例: 自定义线程名字和获取当前线程名

```
class MyThread implements Runnable{
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
```

```

}

MyThread mt=new MyThread();
new Thread(mt,"线程1").start(); // 自定义线程名字
new Thread(mt).start();
new Thread(mt).start();
new Thread(mt,"线程2").start(); // 自定义线程名字
mt.run(); // main

```

结果:

```

线程1
Thread-0
线程2
Thread-1
main

```

如果没有设置线程名字，会自动生成一个不重复的名字。

```

// 匿名线程使用类静态属性自动编号
private static int threadInitNumber;
private static synchronized int nextThreadNum() {
    return threadInitNumber++;
}

```

直接执行 `mt.run()`就是在主方法中调用线程对象的 `run()`方法，获得线程名字为 `main`，所以主方法也是一个线程。

每当使用 `java` 命令执行程序时就启动了一个 `JVM` 的进程，一台电脑可以同时启动多个 `JVM` 进程，一个 `JVM` 进程都有各自的线程。

主线程可以创建若干子线程，主要将一些复杂逻辑或耗时操作交给子线程处理。

```

System.out.println("吃饭");
new Thread(() -> {
    // 模拟耗时操作,耗时操作交给子线程完成
    double pi = 0;
    double flag = 1;
    for (int i = 1; i < 1e9; i += 2) {
        pi += flag / i;
        flag = -flag;
    }
    pi *= 4;
    System.out.println("pi=" + pi);
}).start();
System.out.println("睡觉");

```

主线程复杂整体流程，子线程负责处理耗时操作。

## ② 线程的休眠 (sleep)

```
public static void sleep(Long millis) throws InterruptedException
public static void sleep(Long millis, int nanos) throws InterruptedException
```

休眠时可能产生中断异常 `InterruptedException`，是 `Exception` 的子类，说明该异常必须被处理。

多线程休眠有先后顺序，一个线程休眠会释放执行权，其他线程抢占资源。

```
Runnable run = () -> {
    for (int i = 0; i < 10; i++) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + ", i=" + i);
    }
};
// 5个子线程几乎同时休眠同时唤醒，因为执行太快了；差别是每轮打印顺序都不一样
for (int n = 0; n < 5; n++) {
    new Thread(run, "hikari - " + n).start();
}
```

## ③ 线程中断 (interrupt)

线程休眠可能会产生中断异常，也就是线程休眠可能被其他线程打断。

```
public boolean isInterrupted() // 判断线程是否被中断
public void interrupt() // 中断该线程
```

示例：main 线程中止子线程

```
Thread t = new Thread(() -> {
    System.out.println("该睡觉了...");
    try {
        Thread.sleep(10000); // 预计休眠10s
        System.out.println("醒来...");
    } catch (InterruptedException e) {
        System.out.println("草泥马，不要打扰劳资睡觉!");
    }
});
t.start();
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {}
if (!t.isInterrupted()) {
    // 如果线程没被中止，则中止它；main线程中止子线程
    System.out.println("小伙子，该醒了!");
}
```



```
t.interrupt();
}
```

#### ④ 线程强制执行 (join)

当满足某些条件后，某个线程对象一直独占资源，直到该线程执行结束。

```
public final void join() throws InterruptedException
```

```
public static void main(String[] args) throws Exception {
    // sleep()和join()都会抛出异常，直接在方法上声明吧...
    Thread t = new Thread(() -> {
        for (int x = 0; x < 100; x++) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + ", x=" + x);
        }
    }, "子线程");
    t.start();
    for (int x = 0; x < 100; x++) {
        if (x == 10) {
            t.join(); // 主线程等待主线程t执行完再执行
        }
        Thread.sleep(100);
        System.out.println("【main线程】， x=" + x);
    }
}
```

#### ⑤ 线程的礼让 (yield)

线程的礼让是将资源让给其他线程先执行。

```
public static void yield()
```

```
public static void main(String[] args) throws Exception {
    Thread t = new Thread(() -> {
        for (int x = 0; x < 100; x++) {
            if (x % 5 == 0) {
                // 每次x为5的倍数时，子线程让出执行权
                Thread.yield(); // 静态方法
                System.out.println("*****子线程礼让*****");
            }
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
            }
        }
    });
    t.start();
}
```

```

        e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + ", x=" + x);
}
}, "子线程");
t.start();
for (int x = 0; x < 100; x++) {
    Thread.sleep(100);
    System.out.println("【main线程】, x=" + x);
}
}
}

```

礼让执行每次调用 `yield()` 方法只会礼让一次。该方法很少使用。

## ⑥ 线程优先级

理论上线程的优先级越高越可以先执行(抢占到资源)。

```

public final void setPriority(int newPriority)
public final int getPriority()

```

优先级定义使用 `int`, `Thread` 类定义三个与优先级的 `int` 常量:

```

public static final int MAX_PRIORITY = 10; // 最高优先级
public static final int NORM_PRIORITY = 5; // 中等优先级
public static final int MIN_PRIORITY = 1; // 最低优先级

```

```

public static void main(String[] args) throws Exception {
    System.out.println("主线程优先级: " + Thread.currentThread().getPriority());
    Runnable run = () -> {
        for (int i = 0; i < 100; i++) {
            System.out.println(Thread.currentThread().getName() + " " + i);
        }
    };
    Thread t1 = new Thread(run, "子线程01");
    Thread t2 = new Thread(run, "子线程02");
    Thread t3 = new Thread(run, "子线程03");
    System.out.println("新创建的线程对象的优先级: " + t3.getPriority());
    t1.setPriority(Thread.MAX_PRIORITY); // 将子线程01优先级设为最高
    t1.start();
    t2.start();
    t3.start();
}
}

```

子线程和默认创建的子线程优先级都是 5。

## 20180608

### ✧ 22.8 线程同步

多线程中可以利用 Runnable 描述多线程操作的资源, Thread 描述每个线程对象; 当多个线程访问同一资源时, 就会产生数据操作错误。

```
public class Ticket implements Runnable {
    private int ticket = 5;
    @Override
    public void run() {
        while (true) {
            if (this.ticket > 0) {
                try {
                    Thread.sleep(100); // 模拟网络延时
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + ": ticket
num=" + this.ticket--);
            } else {
                System.out.println("*****票卖完了*****");
                break;
            }
        }
    }
}

Runnable run=new Ticket();
new Thread(run,"票贩子A").start();
new Thread(run,"票贩子B").start();
new Thread(run,"票贩子C").start();
```

比较牛 B 的结果:

```
票贩子 C: ticket num=4
票贩子 B: ticket num=5
票贩子 A: ticket num=3
票贩子 A: ticket num=2
票贩子 B: ticket num=1
*****票卖完了*****
票贩子 C: ticket num=0
*****票卖完了*****
票贩子 A: ticket num=-1
*****票卖完了*****
```

当 ticket=1 时, 票贩子 B 进入 if 判断进行休眠, 因为 ticket 还是 1, 随后票贩子 C 和 A 也进入 if 判断进行休眠; 票贩子 B 休眠结束打印 1, ticket 变为 0; 票贩子 C 休眠结束打印 0, ticket 变为-1; 票贩子 A 休眠结束打印-1, ticket 变为-2。

出现问题的原因是线程休眠前通过 if 判断, 但唤醒时已经不符合条件了。需要将一次卖票的动作同步处理, 一个线程卖票时, 其他线程需要外面等待。

同步是指多个操作在同一时间只能有一个线程进行, 其他线程等待此线程完成

后才能执行。同步的关键是锁，使用**synchronized**关键字可以定义同步方法和同步代码块。

同步代码块：

```
@Override
public void run() {
    while (true) {
        synchronized (this) { // 同步对象一般使用this
            if (this.ticket > 0) { // ... } else { // ... }
        }
    }
}
```

**synchronized** 不要放在 `run()`方法上或把 `while` 放进同步代码块，否则只会有一个线程执行完成后，另外两个再开始已经不满足 `if` 条件，打印后退出。

同步方法：

```
public class Ticket implements Runnable {
    private int ticket = 100;
    private boolean flag = true;
    private synchronized void sell() { // 同步方法
        if (this.ticket > 0) {
            try {
                Thread.sleep(100); // 模拟网络延时
            } catch (InterruptedException e) {}
            System.out.println(Thread.currentThread().getName() + ": ticket
num=" + this.ticket--);
        } else {
            this.flag = false;
            System.out.println("*****票卖完了*****");
        }
    }
}

@Override
public void run() {
    while (this.flag) {
        this.sell();
    }
}
```

系统许多类同步处理都使用同步方法。使用同步会造成程序性能降低。

✧ 22.9 死锁

死锁是多线程同步处理时可能产生的问题，死锁是几个线程互相等待的状态。

无意义的死锁示例：

```
public class DeadLock implements Runnable {
    private static final Robber r = new Robber();
    private static final Person p = new Person();
    private boolean flag = true;
    public DeadLock() {}
    public DeadLock(boolean flag) {
        this.flag = flag;
    }
    @Override
    public void run() {
        if (this.flag) {
            r.say(p); // 一个线程抢占了r，继续执行需要p;
        } else {
            p.say(r); // 另一个线程抢占了p，继续执行需要r，双方一直僵持，互不相让
        }
    }
}

class Robber {
    public synchronized void say(Person p) {
        System.out.println("此山是我开，此树是我栽，要想打此过，留下买路财!");
        p.after();
    }
    public synchronized void after() {
        System.out.println("喽啰们让路!");
    }
}

class Person {
    public synchronized void say(Robber r) {
        System.out.println("先让我走，再给钱!");
        r.after();
    }
    public synchronized void after() {
        System.out.println("别打了，我给钱...");
    }
}

new Thread(new DeadLock()).start();
new Thread(new DeadLock(false)).start();
```

造成死锁主要原因是线程互相等待对方先让出资源。死锁是开发中出现的不确定状态，如果代码处理不当会不定期出现死锁，属于正常开发的调试问题。而示例

是强行死锁，不具备参考性。

#### ✧ 22.10 综合案例：生产者与消费者

多线程开发最著名的案例就是[生产者与消费者](#)：生产者负责信息内容的生产；每当生产者生产一项完整的信息，消费者取走信息。

如果生产者没有生产完成，消费者需要等待后再消费；反之，如果消费者没有处理完成，生产者也需要等待后再生产。

生产者和消费者定义为两个独立的线程对象，定义 `Resource` 类实现两个线程的数据保存。

基本模型：

```
public class Resource {
    private String name;
    private String group;
    public void setName(String name) {
        this.name = name;
    }
    public void setGroup(String group) {
        this.group = group;
    }
    public String toString() {
        return name + ">>>" + group;
    }
}

public class Producer implements Runnable {
    private Resource src;
    public Producer(Resource src) {
        this.src = src;
    }
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
                this.src.setName("星空漂");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {}
                this.src.setGroup("lily white");
            } else {
                this.src.setName("西木野真姬");
                try {
                    Thread.sleep(100);
                }
            }
        }
    }
}
```

```

        } catch (InterruptedException e) {}
        this.src.setGroup("BiBi");
    }
}

}

}

}

public class Consumer implements Runnable {
    private Resource src;
    public Consumer(Resource src) {
        this.src = src;
    }
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {}
            System.out.println(this.src);
        }
    }
}

Resource src = new Resource();
new Thread(new Producer(src)).start();
new Thread(new Consumer(src)).start();

```

不使用同步打印错乱:

```

西木野真姬>>>lily white
星空凛>>>BiBi
西木野真姬>>>lily white
星空凛>>>BiBi
...

```

生产者线程开始设置 name 为 rin，休眠；消费者线程也休眠；生产者线程醒来，group 设为 lily white，此时消费者线程仍在休眠，生产者线程进入下一轮循环，设置 name 为 maki，休眠；消费者线程醒来，打印 maki>>>lily white...

使用同步:

```

public class Resource {
    private String name;
    private String group;
    private int num; // 计数
    public synchronized void set(String name, String group) {
        this.name = name;
        try {

```

```

        Thread.sleep(100);
    } catch (InterruptedException e) {}
    this.group = group;
    num++;
}
@Override
public synchronized String toString() { // 给toString()添加同步没问题?
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {}
    return name + ">>>" + group + ">>>" + num;
}
}

public class Producer implements Runnable {
    // ...
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if (i % 2 == 0) {
                src.set("星空凛", "lily white");
            } else {
                src.set("西木野真姬", "BiBi");
            }
        }
    }
}

public class Consumer implements Runnable {
    // ...
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(this.src);
        }
    }
}

```

打印结果:

```

星空凛>>>lily white>>>3
西木野真姬>>>BiBi>>>12
西木野真姬>>>BiBi>>>14
...
星空凛>>>lily white>>>89
西木野真姬>>>BiBi>>>100
西木野真姬>>>BiBi>>>100
...

```



因为生产者生产得太快了，消费者来不及消费，后期生产者已经完成生产，消费者每次消费都使用最后生产的数据。

#### ✧ 22.11 线程等待与唤醒

解决生产者和消费者问题最好的方法是使用[等待与唤醒机制](#)，主要使用 Object 类几个的方法：

```
public final void wait() throws InterruptedException // 死等
public final void wait(long timeout) throws InterruptedException // 设置等待时间
public final void notify() // 唤醒任意一个等待线程
public final void notifyAll() // 唤醒全部等待线程
```

示例：使用等待唤醒机制

```
public class Resource {
    private String name;
    private String group;
    private int num;
    private boolean flag = true; // true表示允许生产不允许消费
    public synchronized void set(String name, String group) {
        if (!this.flag) { // 无法生产，等待
            try {
                super.wait();
            } catch (InterruptedException e) {}
        }
        this.name = name;
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
        this.group = group;
        num++;
        this.flag = false; // 生产完成
        super.notify();
    }

    @Override
    public synchronized String toString() {
        if (this.flag) { // 正在生产,不能消费
            try {
                super.wait();
            } catch (InterruptedException e) {}
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
        try {
```

```

        return name + ">>>" + group + ">>>" + num;
    } finally { // return之后语句无效,所有需要放到finally代码块
        this.flag = true; // 消费完,可以生成产了
        super.notify();
    }
}
}
}

```

结果:

```

星空凛>>>lily white>>>1
西木野真姬>>>BiBi>>>2
...
星空凛>>>lily white>>>99
西木野真姬>>>BiBi>>>100

```

这是多线程最原始处理方案，整个等待、同步、唤醒机制都是开发者自行通过原生代码实现控制。

#### ✧ 22.12 优雅地停止线程

Thread 类原本提供停止线程的 stop()方法，但从 JDK 1.2 开始就已经废除了，直到现在也不推荐使用。

以下方法已全部废除，原因是可能会导致死锁。

- 1) stop(): 停止线程
- 2) destroy(): 销毁线程
- 3) suspend(): 挂起线程，暂停执行
- 4) resume(): 恢复挂起线程，继续执行

示例：优雅地停止线程

```

public class ElegantStop implements Runnable {
    private boolean flag = true;
    private int num = 0;
    public void setFlag(boolean flag) {
        this.flag = flag;
    }
    @Override
    public void run() {
        while (flag) {
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {}
            System.out.println(Thread.currentThread().getName() + ", num=" +
this.num++);
        }
    }
    public void stop() { // flag设为false, run()方法结束循环，线程结束
        this.setFlag(false);
    }
}

```

```

    }
}

    ElegantStop run = new ElegantStop();
    new Thread(run).start();
    try {
        Thread.sleep(2000); // 主线程休眠2s后
    } catch (InterruptedException e) {}
    run.stop(); // 将子线程停止

```

设置 flag 字段，true 表示线程执行，其他线程修改 flag 为 false，线程停止。

### ✧ 22.13 守护线程

主线程或其他线程在执行时，守护线程将一直存在，并后台运行。如果程序执行完毕，守护线程也就消失。JVM 最大的守护线程是 [GC 线程](#)。

```

public final void setDaemon(boolean on) // 设置为守护线程
public final boolean isDaemon() // 判断是否为守护线程

```

示例：

```

ElegantStop run = new ElegantStop();
ElegantStop daemon = new ElegantStop();
Thread userThread = new Thread(run, "用户线程");
Thread daemonThread = new Thread(daemon, "守护线程");
userThread.start();
daemonThread.setDaemon(true); // 启动之前设置为守护线程
daemonThread.start();

```

如果不停止用户线程，用户线程和守护线程将比翼双飞；如果优雅地停止用户线程，守护线程乘风而去。

### ✧ 22.14 volatile 关键字

多线程中 **volatile** 关键字主要定义属性，表示该属性为直接数据操作，而不进行副本的拷贝处理。一些书错误地理解为同步属性。

正常进行变量处理步骤：

- 1) 获取变量原有的数据内容副本；
- 2) 利用副本为变量进行数学计算；
- 3) 将计算后的变量保存到原始空间中。

如果一个属性定义了 **volatile** 关键字，表示不使用副本，直接操作原始变量，相当于节约了拷贝副本、重新保存的步骤。

面试题： **volatile** 和 **synchronized** 的区别

- 1) **volatile** 主要在属性上使用； **synchronized** 在代码块和方法上使用；
- 2) **volatile** 无法描述同步，只是一种直接内存处理，避免副本的操作； **synchronized** 实现同步。两者没什么联系。

## ✧ 22.15 多线程综合案例：数字加减

设 4 个线程对象，两个执行加操作，两个执行减操作，加减交替执行。

```
public class Resource { // 资源类
    private int num = 0;
    private boolean flag = true; // true为只执行加法,false为只执行减法
    public synchronized void add() {
        while (!this.flag) { // 此处需要while而不是if,否则可能多次加或多次减
            try {
                super.wait(); // 执行减法,加法休眠
            } catch (InterruptedException e) {}
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
        System.out.println(Thread.currentThread().getName() + ", num=" +
++this.num);
        this.flag = !this.flag; // 加法执行完毕,需要执行减法
        super.notifyAll(); // 唤醒全部等待线程,否则可能线程全部休眠
    }

    public synchronized void sub() {
        while (this.flag) {
            try {
                super.wait();
            } catch (InterruptedException e) {}
        }
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {}
        System.out.println(Thread.currentThread().getName() + ", num=" + --
this.num);
        this.flag = !this.flag;
        super.notifyAll();
    }
}

// 加法线程
public class AddThread implements Runnable {
    private Resource src;
    public AddThread(Resource src) {
        this.src = src;
    }
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
```

```

        src.add();
    }
}
// 减法线程
public class SubThread implements Runnable {
    private Resource src;
    public SubThread(Resource src) {
        this.src = src;
    }
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            src.sub();
        }
    }
}

// 测试
Resource src = new Resource();
AddThread add = new AddThread(src);
SubThread sub = new SubThread(src);
new Thread(add, "加法线程1").start();
new Thread(add, "加法线程2").start();
new Thread(sub, "减法线程1").start();
new Thread(sub, "减法线程2").start();

```

结果：加减法交替执行，num 的值为 1 或 0

**注：**此处每种线程有多个，判断 flag 要使用 **while** 而不是 if。否则可能会出现加法之后还是加法或减法之后还是减法。如：

```

t1 --> num=1, flag=false --> notifyAll()
t2 --> wait()
t3 --> num=0, flag=true --> notifyAll()
t1 --> num=1, flag=false --> notifyAll()
t2 --> num=2, flag=false --> notifyAll()
...

```

t2 醒来的时候虽然 flag=false，但之前已经过了 if 判断，还是做加法，num=2，这样就有问题了；所以线程唤醒的时候也需要判断 flag，故用 while。

#### ✧ 22.16 Lock 接口

**synchronized** 对锁的操作的隐式的。JDK 1.5 出现 java.util.concurrent.locks 包，其中的 Lock 接口将 **synchronized** 的隐式操作变为显式的锁操作，使用更加灵活，一个锁可以有多组监视器。

```

void lock() // 获取锁
void unlock() // 释放锁，通常定义在 finally 代码块中

```

ReentrantLock 类是 Lock 接口的常用实现。

Lock 实例的 newCondition() 返回此 Lock 实例绑定的 Condition 实例：

```
Condition newCondition()
```

Condition 接口将对象监视器方法(wait、notify、notifyAll)分解为不同的对象，与任意 Lock 实例组合，使每个对象具有多个等待集的效果。

当 Lock 代替同步方法或同步代码块，要使用 Condition 代替对象监视器方法：

```
void await() throws InterruptedException // 当前线程休眠
void signal() // 唤醒一个等待的线程
void signalAll() // 唤醒所有等待的线程
```

示例：竞猜抢答

设置 3 个抢答者(线程)，同时发出抢答指令，抢答成功显示成功，反之显示失败。

// 线程要返回结果实现Callable

```
class MyThread implements Callable<String> {
    private boolean flag = true;
    private Lock lock = new ReentrantLock(); // 可重入锁
    @Override
    public String call() throws Exception {
        this.lock.lock(); // 获取锁
        try {
            if (this.flag) {
                this.flag = false;
                return Thread.currentThread().getName() + ": 抢答成功!";
            }
            return Thread.currentThread().getName() + ": 抢答失败!";
        } finally {
            this.lock.unlock(); // 释放锁
        }
    }
}
```

```
MyThread mt = new MyThread();
FutureTask<String> task1 = new FutureTask<>(mt);
FutureTask<String> task2 = new FutureTask<>(mt);
FutureTask<String> task3 = new FutureTask<>(mt);
new Thread(task1, "maki").start();
new Thread(task2, "rin").start();
new Thread(task3, "nozomi").start();
System.out.println(task1.get());
System.out.println(task2.get());
System.out.println(task3.get());
```

每次结果不一样:

```
maki: 抢答失败!  
rin: 抢答成功!  
nozomi: 抢答失败!
```

20180609

Ⅱ のーぞーみーちゃんっ！誕生日おめでとうっ♪



希パワー注入♪

## 23 Java 常用类库

### ☆ 23.1 StringBuffer 类

String 类是开发中一定会用到的类，其特点：

- 1) 每个字符串常量都是一个 String 类的匿名对象，内容不可变；
- 2) String 有两个常量池：静态常量池和运行时常量池；
- 3) String 对象实例化建议直接赋值而不是 new。

String 类最大的缺点是对象内容不可修改。StringBuffer 类解决了此问题，可以实现字符串内容的修改。

常用方法：

```
// 1. 构造方法  
public StringBuffer()  
public StringBuffer(String str) // 接收初始化字符串内容  
// 2. 插入数据  
public java.lang.AbstractStringBuilder append(DataType data) // 不能是 byte 或 short  
public StringBuffer insert(int offset, DataType data) // 指定位置插入  
// 3. 删除数据  
public StringBuffer delete(int start, int end)  
// 4. 字符串反转  
public StringBuffer reverse()
```

示例：

```
StringBuffer sb = new StringBuffer();  
sb.append("hello ").append("world").append("!");  
System.out.println(sb.toString()); // hello world!
```

```

sb.delete(6, 11).insert(6, "hikari");
System.out.println(sb.toString()); // hello hikari!
System.out.println(sb.reverse().toString()); // !irakih olleh
System.out.println(sb.toString()); // !irakih olleh

```

JDK 1.5 提供的 `StringBuilder` 类和 `StringBuffer` 类功能相同，但是 `StringBuffer` 的全部使用 **synchronized** 关键字，是线程安全的。

JDK 1.4 提供的 `CharSequence` 接口是描述字符串结构的接口。`String`、`StringBuffer`、`StringBuilder` 是其常用的 3 个子类。

## ✧ 23.2 AutoCloseable 接口

JDK 1.7 提供的 `AutoCloseable` 接口用于实现资源的自动关闭(释放)，如文件、网络、数据库资源的关闭。

该接口只有一个方法：

```
void close() throws Exception
```

实现自动关闭，除了要实现 `AutoCloseable` 接口，还需要结合异常处理语句完成。

```

interface IMessage extends AutoCloseable {
    void send(String msg);
}

class Message implements IMessage {
    public boolean connect() {
        System.out.println("【connect】成功连接!");
        return true;
    }
    @Override
    public void send(String msg) {
        if (this.connect()) {
            System.out.println("发送消息: " + msg);
        }
    }
    @Override
    public void close() throws Exception {
        System.out.println("【close】关闭连接!");
    }
}

// 需要结合异常处理语句完成自动关闭
try (Message m = new Message()) {
    m.send("hello!");
} catch (Exception e) {}

```

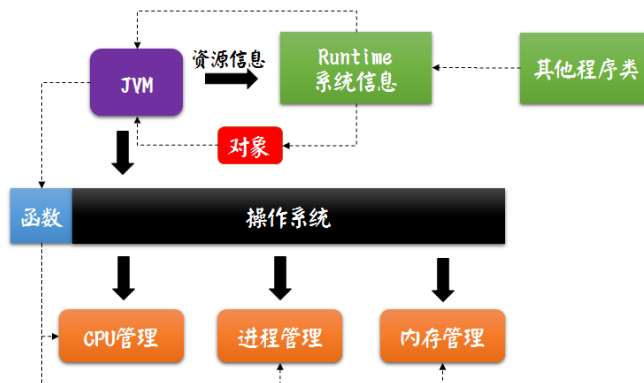


结果:

```
【connect】成功连接!
发送消息: hello!
【close】关闭连接!
```

### ✧ 23.3 Runtime 类

Runtime 类是唯一一个描述 JVM 运行状态的类，默认提供一个实例化对象。每个 JVM 进程只允许提供一个 Runtime 对象，所以其构造方法 private，使用单例设计模式，使用 `getRuntime()` 静态方法获取其实例化对象。



常用方法:

```
public int availableProcessors() // 返回可用 JVM 处理器数量，CPU 内核数
public long maxMemory() // 返回 JVM 可以使用最大内存量
public long totalMemory() // 返回 JVM 中总内存量，值可能随时间变化，默认本机内存 1/4
public long freeMemory() // 返回 JVM 可用内存量，默认本机内存 1/64
public void gc(): // 运行垃圾回收器，手动调用 GC 回收垃圾
```

示例:

```
int MB = 1024 * 1024;
Runtime run = Runtime.getRuntime();
System.out.println("CPU内核数: " + run.availableProcessors()); // 4, CPU内核数
System.out.println(run.maxMemory() / MB + "MB"); // 获取最大可用内存
System.out.println(run.totalMemory() / MB + "MB"); // 总可用内存
System.out.println(run.freeMemory() / MB + "MB"); // 空闲内存空间
String s = "";
// 制造大量垃圾
for (int i = 0; i < 10000; i++) {
    s += i;
}
System.out.println("*****");
System.out.println(run.maxMemory() / MB + "MB");
System.out.println(run.totalMemory() / MB + "MB");
System.out.println(run.freeMemory() / MB + "MB");
run.gc(); // 手动垃圾回收
try {
    Thread.sleep(1000);
}
```

```

} catch (InterruptedException e) {}
System.out.println("*****");
System.out.println(run.maxMemory() / MB + "MB");
System.out.println(run.totalMemory() / MB + "MB");
System.out.println(run.freeMemory() / MB + "MB");

```

结果:

```

CPU内核数: 4
996MB
64MB
62MB
*****
996MB
124MB
119MB
*****
996MB
8MB
7MB

```

## ✧ 23.4 System 类

### ① 数组拷贝

```

public static void arraycopy(Object src, int srcPos, Object dest, int destPos,
int length)

```

示例:

```

int[] arr = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int[] lst = new int[] { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
// arr从下标1开始复制到lst的下标2, 复制长度为5
System.arraycopy(arr, 1, lst, 2, 5);
hikari.ArrayUtil.print(lst); // [9, 8, 1, 2, 3, 4, 5, 2, 1, 0]

```

### ② 返回当前 UTC 时间戳

```

public static Long currentTimeMillis() // 毫秒
public static Long nanoTime() // 纳秒

```

示例: 计算程序运行时间

```

private static void test() {
    String s = "";
    for (int i = 0; i < 100000; i++) {
        s += i;
    }
}

long start = System.currentTimeMillis();
test();
long end = System.currentTimeMillis();
System.out.println(String.format("test()方法用时: %.2fs", (end-
start)/1000.0)); // test()方法用时: 10.21s

```

### ③ 垃圾回收

```
public static void gc()
// System.gc()就是调用 Runtime.getRuntime().gc()
```

#### ✧ 23.5 Cleaner 类

java.lang.ref 包的 Cleaner 类是 JDK 1.9 提供的一个对象清理操作，主要功能是 finalize()方法的替代。

C++有两种特殊函数：构造函数、析构函数(对象手动回收)。

Java 所有垃圾都是 GC 自动回收，所以很多时候不需要析构函数。

但是 Java 提供了给对象收尾的操作(临终遗言)

Object 类的 finalize()方法：

```
@Deprecated(since="9")
protected void finalize() throws Throwable
```

最大特点是抛出 Throwable 类型，它是 Error 和 Exception 的直接父类。

以前 finalize 的使用：

```
class Person {
    public Person() {
        System.out.println("【构造】在一个雷电交加的夜晚诞生了...");
    }
    @Override
    protected void finalize() throws Throwable {
        try {
            System.out.println("【析构】人终有一死，或重于泰山，或重于另一座山...");
            throw new Exception("我真的还想再活500年!");
        }finally {
            super.finalize(); // 已过时
        }
    }
}

Person p = new Person();
p = null; // 成为垃圾
System.gc(); // 手动回收打印遗言
```

结果：

```
【构造】在一个雷电交加的夜晚诞生了...
【析构】人终有一死，或重于泰山，或重于另一座山...
```

Finalization 可能导致性能问题、死锁、和挂起，finalizers 的错误可能导致资源泄露。占用非堆资源的实例的类需要提供一个方法来显式释放资源，按需实现 AutoCloseable 接口。当一个对象不可访问时，Cleaner 和 PhantomReference 提供更加灵活有效的方法释放资源。

```
import java.lang.ref.Cleaner;
public class CleaningExample implements AutoCloseable {
```

```

// 最好是库中共享的cleaner
private static final Cleaner cleaner = Cleaner.create();
static class State implements Runnable { // 内部类
    State() {
        // 初始化清理动作所需的状态
        System.out.println("【构造】在一个雷电交加的夜晚诞生了...");
    }
    public void run() { // 清理使用多线程
        // 清除操作访问状态，最多执行一次
        System.out.println("【析构】人终有一死，或重于泰山，或重于另一座山...");
    }
}
private final State state;
private final Cleaner.Cleanable cleanable;
public CleaningExample() {
    this.state = new State();
    this.cleanable = cleaner.register(this, state);
}
public void close() {
    cleanable.clean(); // 启动多线程
}
}

try (CleaningExample clean = new CleaningExample()) {
    // ...
} catch (Exception e) {}

```

新一代的清除回收更多考虑的是多线程的使用，为了防止有可能造成的延迟，许多对象回收前的处理都是单独通过一个线程完成，提供性能。

## ✧ 23.6 对象克隆

Object 类的 clone() 方法

```
protected Object clone() throws CloneNotSupportedException
```

所有的类都继承 Object，所有的类一定有 clone() 方法，但不是所有类都希望被克隆。如果要想实现对象克隆，对象的类需要实现 Cloneable 接口，否则抛出异常。此接口没有一个方法，其主要描述的是一种能力。

```

class Person implements Cloneable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override

```

```

    public String toString() {
        return "name=" + name + ", age=" + age;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

Person p1 = new Person("hikari", 25);
try {
    Person p2 = (Person) p1.clone();
    // 克隆后的对象内容一样，内存地址不一样
    System.out.println(p1.toString().equals(p2.toString())); // true
    System.out.println(Integer.toHexString(p1.hashCode())); // 299a06ac
    System.out.println(Integer.toHexString(p2.hashCode())); // 383534aa
} catch (CloneNotSupportedException e) {}

```

实际开发很少需要对象的克隆。

### ✧ 23.7 Math 类

Math 类构造方法私有化，所有方法都是 static，静态导入比较合适。

```

System.out.println(Math.PI); // 3.141592653589793
System.out.println(Math.pow(2, 3)); // 8.0
System.out.println(Math.round(-1.5)); // -1, +0.5 后向下取整

```

示例：自定义四舍五入：

```

class MyMath {
    private MyMath() {}
    public static double round(double n, int scale) {
        double x = Math.pow(10, scale);
        return Math.round(n * x) / x;
    }
}

// -1.235*100结果是-123.50000000000001, 四舍五入为-124...
System.out.println(MyMath.round(-1.235, 2)); // -1.24

```

### ✧ 23.8 Random 类 (java.util 包)

一个重要方法是获取不超过某个值的随机非负整数：

```

public int nextInt(int bound) // 产生[0, bound)的随机整数

```

示例：模拟 36 选 7

```

private static ArrayList<Integer> select(int num, int total) {
    ArrayList<Integer> arr = new ArrayList<>();
    Random r = new Random();
}

```

```

    for (int i = 0; i < num; i++) {
        int n = r.nextInt(total);
        if (!arr.contains(n) && n != 0) { // 不重复,不包含0
            arr.add(n);
            i++;
        }
    }
    Collections.sort(arr); // 排序
    return arr;
}

System.out.println(select(7, 36)); // [9, 13, 14, 23, 28, 32, 33]

```

## 20180610

### ✧ 23.9 BigInteger 和 BigDecimal

实现海量数字的基础计算,比如超过 double 范围的数字计算。最早只能使用 String 类保存,但是计算需要逐位拆分,十分麻烦。java.math 包的 BigInteger 和 BigDecimal 是大数字类,是抽象类 Number 的子类。

示例: BigInteger 的运算

```

String sa = "9876543210";
String sb = "1234567890";
// 构造大数字的字符串形式
for (int i = 0; i < 6; i++) {
    sa += sa;
    sb += sb;
}
System.out.println("len(sa)=" + sa.length() + ", len(sb)=" + sb.length());
BigInteger a = new BigInteger(sa);
BigInteger b = new BigInteger(sb);
// 四则运算
System.out.println("a+b=" + a.add(b));
System.out.println("a-b=" + a.subtract(b));
System.out.println("a*b=" + a.multiply(b));
System.out.println("a/b=" + a.divide(b));
// 求商和余数, 结果是两个BigInteger的数组
BigInteger[] ret = a.divideAndRemainder(b);
System.out.println("a/b的商: " + ret[0] + ", 余数: " + ret[1]);
// 计算指数会非常慢
System.out.println(a.pow(Integer.MAX_VALUE));

```

如果数据没有超过基本数据类型, 不要使用大数字类型, 计算性能太差。

BigDecimal 使用和 BigInteger 类似, 但其除法有个数据进位的问题。

```
public BigDecimal divide(BigDecimal divisor, int scale, RoundingMode
roundingMode)
```

示例：使用 BigDecimal 四舍五入：

```
private static double round(double n, int scale) {
    // n转为BigDecimal除以BigDecimal的1, 四舍五入, 转为double
    return new BigDecimal(n).divide(BigDecimal.ONE, scale,
RoundingMode.HALF_UP).doubleValue();
}

System.out.println(round(-1.235, 2)); // -1.24
```

## ✧ 23.10 Date 类

简单 Java 类的设计来源于数据表的结构，日期是数据表常用的类型。

Java 提供 java.util.Date 类处理日期。

```
Date date = new Date();
System.out.println(date); // Sun Jun 10 22:44:35 JST 2018
```

其构造方法：

```
public Date() { // 无参构造默认使用当前毫秒
    this(System.currentTimeMillis());
}
public Date(Long date) {
    fastTime = date;
}
```

Date 类只是对 long 的一种包装，Date 类提供日期与 long 之间的转换。

```
public Date(Long date) // long 毫秒值转为 Date 对象
public Long getTime() // Date 对象转为 UTC 时间戳(long 类型的毫秒值)
```

示例：

```
long now = new Date().getTime();
long afterTenDays = now + 8 * 24 * 60 * 60 * 1000;
System.out.println(new Date(afterTenDays)); // Mon Jun 18 22:49:27 JST 2018
```

## ✧ 23.11 SimpleDateFormat 类

Date 默认打印的结构并不十分友好，需要对日期格式化处理。

java.text 包的 SimpleDateFormat 类用于以本地敏感的方式格式化和解析日期。

SimpleDateFormat 继承于抽象类 DateFormat，后者继承于抽象类 Format。

常用方法：

```
public SimpleDateFormat(String pattern) // 构造方法，定义日期格式
public final String format(Date date) // DateFormat 提供，Date 对象转为字符串
public Date parse(String source) throws ParseException // DateFormat 提供，字符串转为 Date 对象
```

示例:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
// 1. Date对象转为字符串
Date date = new Date();
System.out.println(sdf.format(date)); // 2018-06-10 23:12:58
// 2. 字符串转为Date对象
String someday = "2020-02-02 20:02:20";
try {
    Date day = sdf.parse(someday);
    System.out.println(day); // Sun Feb 02 20:02:20 JST 2020
} catch (ParseException e) {}
// 附赠: 数字格式化
NumberFormat nf = NumberFormat.getInstance();
long n = 1234567890;
System.out.println(nf.format(n)); // 1,234,567,890
```

如果字符串使用的日期超过了合理范围, 则会自动进位。

## 24 正则表达式

### ✧ 24.1 正则表达式简介

String 是非常万能的类型, 因为支持向各种数据类型转换。开发中用户输入的信息基本都是 String。String 向其他数据类型转换时, 可能需要进行复杂的验证。比如验证字符串是否全是数字组成, 之前的练习将字符串转为字符数组, 逐个比较, 代码略麻烦。使用正则表达式验证则可简化代码。

```
String s = "123";
if (s.matches("\\d+")) {
    int n = Integer.parseInt(s);
    System.out.println(n + "*" + n + "=" + n * n); // 123*123=15129
}
```

正则表达式最早从 Perl 语言里发展而来, JDK 1.4 后支持正则表达式, 提供 java.util.regex 包, 同时 String 也支持正则表达式。

正则表达式主要特点是方便验证处理, 复杂字符串修改或获取指定数据。

### ✧ 24.2 常用正则表达式结构

#### ① 字符类

|          |                      |
|----------|----------------------|
| [abc]    | a 或 b 或 c            |
| [^abc]   | 除了 a, b, c 的任何字符(否定) |
| [a-zA-Z] | a 到 z 或 A 到 Z(范围)    |

#### ② 预定义的字符类

|    |                      |
|----|----------------------|
| .  | 任意字符 (行结束符可能(不)匹配)   |
| \d | 数字: [0-9]            |
| \D | 非数字: [^0-9]          |
| \s | 空白字符: [\t\n\x0B\f\r] |



```
\S 非空白字符: [^\s]
\w 单词字符: [a-zA-Z_0-9]
\W 非单词字符: [^\w]
```

### ③ 匹配边界

```
^ 一行开始
$ 一行结束
\b 单词边界
\B 非单词边界
```

### ④ 数量词(贪婪模式)

```
X?      X, 一次或零次
X*      X, 零次或多次
X+      X, 一次或多次
X{n}    X, 恰好 n 次
X{n,}   X, 至少 n 次
X{n,m}  X, 至少 n 次, 至多 m 次
```

默认使用贪婪模式匹配, 尽可能匹配多的字符。

数量词后面加?表示非贪婪模式, 尽可能匹配少的字符。

### ⑤ 逻辑运算符

```
XY      XY 紧随
X|Y     X 或 Y
(X)     X, 作为一个捕获组
```

### ⑥ 反向引用

```
\n      任何匹配的 nth 捕获组
\k<name> 任何匹配的命名捕获组 "name"
```

### ⑥ 特殊结构 (命名捕获和非捕获)

```
(?<name>X)  X, 作为命名捕获组
(?:X)     X, 作为非捕获组
```

捕获组可以通过左括号的顺序来编号。

如表达式((A)(B(C)))中有四个这样的组:

1. ((A)(B(C)))
2. (A)
3. (B(C))
4. (C)

0 组始终代表整个表达式。

## ✧ 24.3 String 对正则表达式的支持

正则表达式有单独的包 `java.util.regex` 包, 有两个主要的类 `Matcher` 和 `Pattern`, 如果不是必要, 此包很少用, 因为 `String` 也支持正则表达式, 可以直接使用。

String 类关于正则表达式的常用方法:

```
public boolean matches(String regex)
public String replaceFirst(String regex, String replacement)
public String replaceAll(String regex, String replacement)
public String[] split(String regex)
public String[] split(String regex, int limit)
```

练习: 验证 email 格式

用户名由字母、数字、\_组成, 不能\_开头; 域名由字母、数字组成; 域名后缀必须是: .cn、.com、.net、.com.cn、.org

```
String email = "hikari@qq.com";
String re = "[a-zA-Z0-9]\\w+@[a-zA-Z0-9]+\\.\\.(cn|com|net|com\\.cn|org)";
System.out.println(email.matches(re) ? "匹配成功!" : "匹配失败!"); // 匹配成功!
```

示例: 使用捕获组

```
String s = "umi,rinrinrin,makimaki,nozominozomi,nico";
// ()为捕获组,\1取第一组,此处表示一组字符连续重复出现
String re = "([a-z]+)\\1+";
// "$"获取前一个正则参数的内容, "$1"为获取第一组
s = s.replaceAll(re, "$1");
System.out.println(s); // umi,rin,maki,nozomi,nico
```

#### ✧ 24.4 java.util.regex 包

有两个主要的类 Matcher 和 Pattern。

##### ① Pattern

构造方法私有, 提供正则表达式编译处理。

```
public static Pattern compile(String regex)
```

提供 split()方法, 功能与字符串的 split()一样

```
public String[] split(CharSequence input)
```

##### ② Matcher

实现正则表达式匹配, 实例化依靠 Pattern 类的 matcher()方法

```
public Matcher matcher(CharSequence input)
```

Matcher 对象主要方法:

```
public boolean matches()
public String replaceFirst(String replacement)
public String replaceAll(String replacement)
```

如果只是 split、replace、matches 根本不需要使用 java.util.regex 包, 直接使用 String 的方法即可。

Matcher 类提供分组功能, 是 String 类不具备的。

```
public boolean find() // 试图找到匹配的下一个子序列
public String group() // 返回前一个匹配的子序列
public String group(int group) // 返回上次匹配操作中被给定组捕获的子序列
```

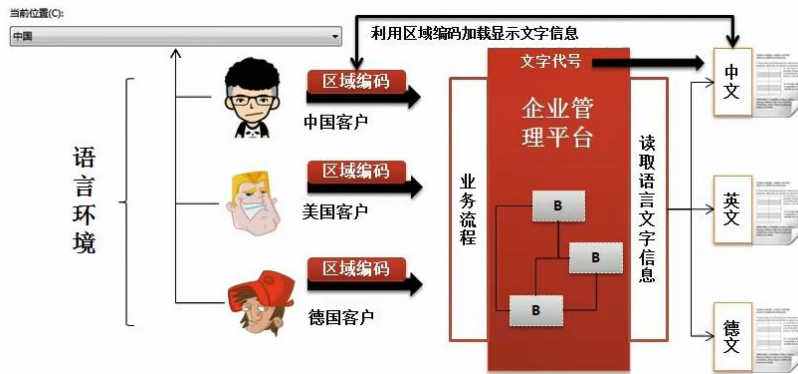
示例:

```
String html = "<ul><li>rin</li><li>maki</li><li>nozomi</li></ul>";
String re = "<li>([a-z]+)</li>";
Pattern p = Pattern.compile(re);
Matcher m = p.matcher(html);
while (m.find()) { // 是否有匹配内容
    System.out.println(m.group(1));
}
```

20180611

## 25 国际化程序

国际化程序是同一个程序代码可以根据不同国家实现不同的语言描述,但是程序的核心业务的相同的。



要实现国际化开发, 需要解决的问题:

- 1) 如何定义保存文字的文件信息;
- 2) 如何根据不同的区域语言的编码读取指定的资源信息。

### ✧ 25.1 Locale 类

java.util 包的 Locale 类专门描述区域和语言编码。

```
public Locale(String language)
public Locale(String language, String country)
```

语言和国家的代码, zh-CN: 中文(简体); en\_US: 英语(美国)。

```
// 手动选择
Locale loc = new Locale("zh", "CN");
System.out.println(loc); // zh_CN
```

Locale 类默认方式实例化可以自动获取当前的运行环境。

```
// 读取本地默认环境
Locale local = Locale.getDefault();
System.out.println(local); // zh_CN
```

然而实际开发中, 很多人不关系国家和语言的编码。为了简化开发, Locale 类将世界上一些著名的国家编码设为常量。

```
Locale jp = Locale.JAPAN;
System.out.println(jp); // ja_JP
Locale en = Locale.UK;
System.out.println(en); // en_GB
```

## ✧ 25.2 配置资源文件

国际化程序中，语言文字最为重要。可以将所有语言文字保存在资源文件中：

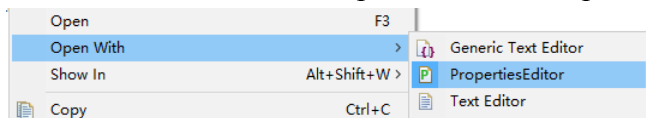
- 1) 所有资源文件一定要定义在 CLASSPATH 中，允许资源文件在包里；
- 2) 资源文件的后缀必须是 .properties，也可以成为属性文件；
- 3) 资源文件所有数据使用字符串 key=value 形式定义。

资源文件命名使用"资源文件名\_语言\_国家.properties"

注意：.properties 保存中文需要转码，原本 JDK 有专门的转码工具，但是 JDK 1.9 后就消失了。现在直接利用 Eclipse 处理，但是中文显示 Unicode 编码，不方便维护。需要在 Eclipse 安装一个资源文件的编辑插件 [PropertiesEditor](#)。

Help → Install New Software → Add → Name:properties,Location:将链接复制 → 联网加载 → 下载新版本 PropertiesEditor → Next → Next → accept 同意协议 → Finish → Install anyway 忽略警告 → Restart Now 重启 Eclipse

安装完成后：右击配置文件 Open With → PropertiesEditor 打开



## ✧ 25.3 读取资源文件(ResourceBundle 类)

读取资源文件主要依靠 java.util.ResourceBundle 类完成，此类是一个抽象类。但可以直接使用静态方法 getBundle() 获取其实例化对象。

```
public static final ResourceBundle getBundle(String baseName)
public static final ResourceBundle getBundle(String baseName, Locale locale)
```

baseName 是资源文件的名称，没有后缀

示例：

```
// 资源文件在包里,需要带包的名字
ResourceBundle src = ResourceBundle.getBundle("java0611.data");
String name = src.getString("name");
String group = src.getString("group");
int age = Integer.parseInt(src.getString("age"));
System.out.println("name:" + name + ", group:" + group + ", age:" + age);
// name:星空凛, group:lily white, age:15
```

如果资源文件不存在，抛出：

```
Exception in thread "main" java.util.MissingResourceException: Can't find bundle for base
name data, locale zh_CN
```

如果访问的 key 不存在，抛出：

```
Exception in thread "main" java.util.MissingResourceException: Can't find resource for
bundle java.util.PropertyResourceBundle, key xxx
```

## ✧ 25.4 实现国际化程序

写几个资源文件：中文、英文、日文、没有区域的资源文件

```
message_en_US.properties
message_ja_JP.properties
message_zh_CN.properties
message.properties
```

```
// 没有指定Locale对象,默认使用本地默认Locale.getDefault()加载中文资源
ResourceBundle src = ResourceBundle.getBundle("java0611.message");
String info = src.getString("info");
System.out.println(info); // 很高兴见到你!
// 指定地区为Japan加载日文资源
ResourceBundle srcJP = ResourceBundle.getBundle("java0611.message",
Locale.JAPAN);
System.out.println(srcJP.getString("info"));
// はじめまして、どうぞよろしくお願いします!
```

读取顺序：指定区域的资源 > 默认本地资源 > 公共资源(没有设置区域)

## ✧ 25.5 消息格式化

修改资源文件：

```
info={0}, 很高兴见到你! 今天是{1}.
```

{0}、{1}之类是占位符，可以继续往后添加。

如果直接读取，会将占位符读取出来，需要 java.text.MessageFormat 类格式化处理。其提供一个格式化文本的方法：

```
public static String format(String pattern, Object... arguments)
```

示例：

```
// 没有指定Locale对象,默认使用本地默认Locale.getDefault()加载中文资源
ResourceBundle src = ResourceBundle.getBundle("java0611.message");
String info = src.getString("info");
Date today = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日");
info = MessageFormat.format(info, "hikari", sdf.format(today));
// 格式化处理
System.out.println(info); // hikari, 很高兴见到你! 今天是2018年06月11日.
```

## 26 开发支持类

### ✧ 26.1 java.util.Arrays 类

全部是静态方法，构造方法私有化。

常用方法：

① 转为字符串

```
public static String toString(DataType[] a)
```

② 数组排序

```
public static void sort(DataType[] a)
```

③ 二分查找(需要先升序排序)

```
public static int binarySearch(DataType[] a, DataType key)
```

示例:

```
int[] arr = new int[] { 9, 8, 7, 6, 5, 4, 3, 2, 1 };
Arrays.sort(arr);
System.out.println(Arrays.toString(arr)); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
// 二分查找前需要升序排序,否则找不到返回-1
System.out.println(Arrays.binarySearch(arr, 2)); // 1
```

④ 数组比较

```
public static int compare(DataType [] a, DataType [] b)
```

返回 1(大于)、-1(小于)或 0(等于)。

⑤ 数组相等判断

```
public static boolean equals(DataType[] a, DataType[] a2)
```

⑥ 数组填充

```
public static void fill(DataType[] a, DataType val)
```

示例:

```
int[] arr = new int[10];
// arr全部填充1
Arrays.fill(arr, 1);
System.out.println(Arrays.toString(arr)); // [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

虽然说这些功能 Arrays 类都实现好了,但是排序和二分查找仍是面试常考题,需要会写,实际开发几乎不需要自己实现。

示例: 二分查找

```
public static int binarySearch(int[] arr, int key) {
    if (arr == null) {
        return -1;
    }
    return binarySearch0(arr, 0, arr.length, key);
}

private static int binarySearch0(int[] arr, int from, int to, int key) {
    int low = from;
    int high = to - 1;
    while (low <= high) {
        // 位运算优势是速度更快,但是在内存情况在提升有限
        int mid = (low + high) >>> 1;
        if (arr[mid] == key) {
```

```

        return mid;
    } else if (arr[mid] > key) {
        high = mid - 1;
    } else {
        low = mid + 1;
    }
}
return -1;
}

```

## ✧ 26.2 java.util.UUID 类

UUID (universally unique identifier)是通用唯一标识符，主要功能是根据时间戳生成一个自动的无重复字符串，占 128bit。

一般获取 UUID 对象使用：

```
public static UUID randomUUID()
```

根据字符串创建 UUID 对象：

```
public static UUID fromString(String name)
```

示例：

```

UUID uid = UUID.randomUUID();
String s = uid.toString();
System.out.println(s); // ce9202a6-eb96-463f-939c-266e89fc40d5
System.out.println(s.length()); // 36

```

UUID 为 128bit，32 位 16 进制字符串，所以-不算吗？

UUID 一般用于对一些文件自动命名的处理。

## ✧ 26.3 java.util.Optional<T>类

Optional 类(Since 1.8)主要进行 null 相关处理。为了防止出现 NullPointerException 一般需要 if 判断，也就是接收的一方需要被动的进行判断。

为了解决此种被动，Java 提供 Optional 类可以实现 null 的处理。

主要方法：

```

1. public static <T> Optional<T> empty() // 返回空的 Optional 实例
2. public T get() // 如果 value 存在返回 value，否则抛出 NoSuchElementException
3. public static <T> Optional<T> of(T value) // 将非空 value 的 Optional 实例返回，
   value 为 null 抛出 NullPointerException
4. public static <T> Optional<T> ofNullable(T value) // value 允许为空
5. public T orElse(T other) // 如果 value 存在返回 value，否则返回 other

```

无意义的示例：of()和 get()使用

```

interface IMessage {
    String getContent();
}

```

```

class Message implements IMessage {
    @Override
    public String getContent() {
        return "makirin";
    }
}

class MessageUtil {
    private MessageUtil() {}
    public static Optional<IMessage> getMessage() {
        // return Optional.of(new Message());
        // 将Message对象用Optional类包装，传入null在此处报错，而不是在使用处
        return Optional.of(null);
    }
    public static void useMessage(Optional<IMessage> op) {
        IMessage msg = op.get();
        System.out.println(msg.getContent());
    }
}

Optional<IMessage> op = MessageUtil.getMessage();
MessageUtil.useMessage(op);

```

无意义的示例：使用 ofNullable()和 orElse()

```

class MessageUtil {
    private MessageUtil() {}
    public static Optional<IMessage> getMessage() {
        // ofNullable() 允许传入null但获取时应该使用orElse()并且给个默认值
        return Optional.ofNullable(null);
    }
    public static void useMessage(Optional<IMessage> op) {
        // 当Optional对象中value为null时使用orElse()提供的值
        IMessage msg = op.orElse(new Message());
        System.out.println(msg.getContent());
    }
}

```

所以 Optional 类到底有什么好处呢？个人认为 Optional 类是对其他对象的包装类，通过 ofNullable()和 orElse()结合使用，允许 value 有 null，但提供一个默认值，如此可以有效地规避空指针异常。

#### ✧ 26.4 java.lang.ThreadLocal<T>类

提供线程局部变量。ThreadLocal 变量活动范围只局限于某个线程，也就是每个线程独享一个 ThreadLocal 变量，而不是共享。



```

1. public ThreadLocal() // 构造方法
2. public void set(T value) // 设置数据
3. public T get() // 获取数据
4. public void remove() // 删除数据

```

ThreadLocal 的 set()和 get()方法代码:

```

public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}

public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

```

根据线程对象 t 获取 ThreadLocalMap 对象 map 的 getMap()方法:

```

ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

```

Thread 类的 threadLocals 属性;

```

/* 与此线程相关的 ThreadLocal 值，该映射由 ThreadLocal 类维护 */
ThreadLocal.ThreadLocalMap threadLocals = null;

```

ThreadLocalMap 类是定义在 ThreadLocal 类的静态内部类。Entry 是定义在 ThreadLocalMap 的静态内部类。此 Entry 用于存储键值对。

```

static class Entry extends WeakReference<ThreadLocal<?>> {
    /** 与此 ThreadLocal 关联的值 */
    Object value;
    Entry(ThreadLocal<?> k, Object v) {

```

```

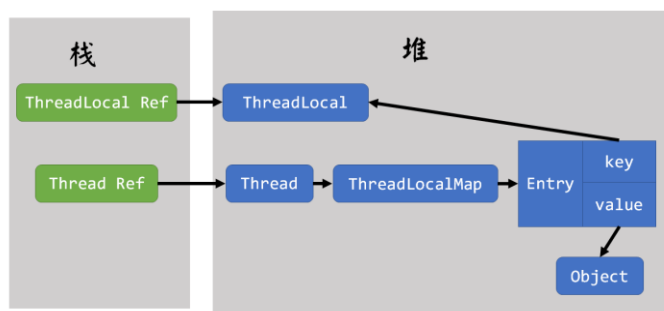
    super(k);
    value = v;
}
}

```

ThreadLocal 实例为 key，用户传入的值为 value。

set(): Thread Ref→Thread→ThreadLocalMap→Entry, Entry.value=value

get(): Thread Ref→Thread→ThreadLocalMap→Entry, return Entry.value



示例：不使用 ThreadLocal

```

class Message {
    private String info;
    public Message(String info) {
        this.info = info;
    }
    public String getInfo() {
        return info;
    }
}

class Channel {
    private static Message msg;
    private Channel() {}
    public static void setMsg(Message msg) {
        Channel.msg = msg;
    }
    public static void send() {
        System.out.println "[" + Thread.currentThread().getName() + " ] " +
msg.getInfo());
    }
}

class MyThread implements Runnable {
    private Message msg;
    public MyThread(String info) {
        this.msg = new Message(info);
    }
}

```

```

@Override
public void run() {
    Channel.setMsg(this.msg);
    Channel.send();
}
}

for (int i = 1; i < 7; i++) {
    new Thread(new MyThread("第" + i + "个线程的消息"), "线程" +
i).start();
}

```

结果:

```

[线程 2] 第 4 个线程的消息
[线程 4] 第 4 个线程的消息
[线程 1] 第 1 个线程的消息
[线程 3] 第 3 个线程的消息
[线程 6] 第 6 个线程的消息
[线程 5] 第 4 个线程的消息

```

多个线程同时发送消息，不同线程之间产生了影响，因为不同步，不同线程 Channel 的 msg 数据产生了覆盖。

在保持核心结构不改变的前提下，每个线程需要独享数据。

示例：使用 ThreadLocal 类为每个线程保存独立数据

```

class Channel {
    private static final ThreadLocal<Message> TL = new ThreadLocal<>();
    private Channel() {}
    public static void setMsg(Message msg) {
        TL.set(msg); // 向ThreadLocal保存数据
    }
    public static void send() { // 从ThreadLocal获取数据
        System.out.println "[" + Thread.currentThread().getName() + " ] " +
TL.get().getInfo());
    }
}

```

结果:

```

[线程 1] 第 1 个线程的消息
[线程 2] 第 2 个线程的消息
[线程 6] 第 6 个线程的消息
[线程 4] 第 4 个线程的消息
[线程 3] 第 3 个线程的消息
[线程 5] 第 5 个线程的消息

```

## 20180612

### ✧ 26.5 定时器

Java 支持定时任务，但是只是实现了间隔触发的操作。

java.util.TimerTask 类：实现 Runnable 的抽象类，用于定义任务主体；

java.util.Timer 类：进行任务启动

```
public void schedule(TimerTask task, Long delay) // 延迟 delay 毫秒任务启动
public void scheduleAtFixedRate(TimerTask task, Long delay, Long period) // delay
毫秒后开始执行，每 period 毫秒执行 1 次
```

示例：

```
class MyTask extends TimerTask {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": 定时任务执
行...");
    }
}

Timer timer = new Timer();
// delay=0表示立即执行
timer.schedule(new MyTask(), 1000);
```

然而任务启动了，只执行一次，貌似一直在运行，需要手动 Terminate

```
class MyTask extends TimerTask {
    private int cnt = 0;
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + ": 定时任务执
行...cnt=" + ++cnt);
    }
}

Timer timer = new Timer();
// 1s后开始执行任务,每2s执行一次任务(run()方法)
timer.scheduleAtFixedRate(new MyTask(), 1000, 2000);
```

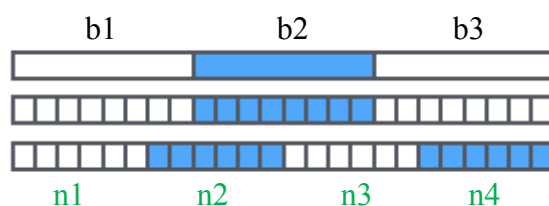
这种定时是 JDK 提供最原始方式，实际开发如果使用此类方式实现定时处理，代码会非常麻烦。

#### ✧ 26.6 java.util.Base64 (Since 1.8)

base64 原理是先准备一个 64 个字符的数组：

{'A', 'B', 'C' ... 'a', 'b', 'c' ... '0', '1' ... '+', '/'}

用 64 个字符表示任意二进制数据。每 3 个字节 1 组，为 24bit，再分为 4 组，每组刚好 6bit。



Base64 编码后长度增加 33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。如果编码的二进制数据不是 3 的倍数，用 \x00 字节在末尾补足，再在编码的末尾加上 1 个或 2 个 = 号，表示补了多少字节，解码时会自动去除。

Base64 类中有两个静态内部类：

```
public static class Base64.Encoder // Base64 编码内部类
|- public byte[] encode(byte[] src) // Base64 编码方法，得到字节数组
|- public String encodeToString(byte[] src) // 得到字符串
public static class Base64.Decoder // Base64 解码内部类
|- public byte[] decode(byte[] src) // Base64 解码方法，参数可以是字节数组或字符串
|- public byte[] decode(String src)
public static Base64.Encoder getEncoder() // 获取 Base64 编码内部类实例
public static Base64.Decoder getDecoder() // 获取 Base64 解码内部类实例
```

示例：

```
// 会使用默认编码将字符串转为字节数组，所以编码设为utf-8吧
String bs = Base64.getEncoder().encodeToString(s.getBytes());
System.out.println(bs); // 5pif56m65Yeb
String s1 = new String(Base64.getDecoder().decode(bs));
System.out.println(s1); // 星空凜
```

Base64 是一种通过查表的编码方法，不能用于加密。适用于小段内容的编码，如数字证书签名、Cookie 内容等。

示例：没卵用的 Base64 撒盐+多次编码和解码

```
private static final String SALT = "<hoshizora rin>"; // 撒盐字符串
private static final int REPEAT = 5; // 重复次数
public static String encode(String s) {
    for (int i = 0; i < REPEAT; i++) {
        s = Base64.getEncoder().encodeToString((s + SALT).getBytes());
    }
    return s;
}
public static String decode(String s) {
    for (int i = 0; i < REPEAT; i++) {
        String tmp = new String(Base64.getDecoder().decode(s));
        s = tmp.replaceAll(SALT, "");
    }
    return s;
}
```

## 27 比较器

### ✧ 27.1 比较器的引出

比较器就是大小关系的判断比较。

Arrays.sort()可以接收一个对象数组将其排序

```
public static void sort(Object[] a)
```

示例：String 数组和 Integer 数组的排序

```
// Arrays.sort()接收String数组和Integer数组
String[] arr = new String[] { "abc", "acb", "bac", "aBC" };
Arrays.sort(arr);
System.out.println(Arrays.toString(arr)); // [aBC, abc, acb, bac]
Integer[] arr1 = new Integer[] { 6, 5, 7, 9, 2 };
Arrays.sort(arr1);
System.out.println(Arrays.toString(arr1)); // [2, 5, 6, 7, 9]
```

如果是自定义类型呢？比如 Person 类：

```
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "<name:" + name + ", age:" + age + ">";
    }
}

Person[] arr = new Person[] { new Person("kotori", 16), new
Person("maki", 15), new Person("nozomi", 17) };
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
```

抛出运行时异常：

```
Exception in thread "main" java.lang.ClassCastException: java0612.Person cannot
be cast to java.base/java.lang.Comparable
```

说明任意的类默认无法实现数组排序或比较，因为没有明确如何比较。

Java 为了统一比较规则，提供比较器的 Comparable 接口。

## ✧ 27.2 Interface Comparable<T>

Comparable 接口实现比较的规则，其中只有一个 compareTo()方法

```
int compareTo(T o)
```

String、Integer 等类实现了 Comparable 接口，所以可以排序。

对象数组排序不关心对象是什么类型，只关心有没有实现 Comparable 接口和有没有覆写 compareTo()方法。没有实现 Comparable 接口的类无法实现数组排序或比较。

示例：Person 类实现 Comparable 接口

```
class Person implements Comparable<Person> {  
    // ...  
    @Override  
    public int compareTo(Person o) {  
        return this.age - o.age;  
    }  
}
```

结果：

```
[<name:maki, age:15>, <name:kotori, age:16>, <name:nozomi, age:17>]
```

### ✧ 27.3 Interface **Comparator**<T>

Comparator 接口属于挽救的比较器功能，主要目的是解决没有实现 Comparable 的类对象的数组排序问题。

```
@FunctionalInterface  
public interface Comparator<T>
```

Comparator 定义为函数式接口，主要有 compare()方法

```
int compare(T o1, T o2)
```

JDK 1.8 新增了一些 static 和 default 方法。

如项目开发完了，但之前设计并没有考虑到需要比较。后来若干个版本后，发现需要有排序功能，但不能修改原来的类(无法实现 Comparable)，需要使用挽救的方法。Arrays 类有一个基于 Comparator 的排序：

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

第二个参数是 Comparator 接口子类实例，属于挽救规则；Comparator 接口子类需要覆写 compare()方法。

```
class PersonComparator implements Comparator<Person> { // 挽救规则  
    @Override  
    public int compare(Person o1, Person o2) {  
        return o1.getAge() - o2.getAge();  
    }  
}  
  
Arrays.sort(arr, new PersonComparator());
```

也可以直接使用 Lambda 表达式：

```
Arrays.sort(arr, (p1, p2) -> p1.getAge() - p2.getAge());
```

不建议使用 Comparator，最好以 Comparable 为主。

面试题：Comparable 和 Comparator 的区别

- 1) java.lang.Comparable 是类定义时实现的父接口，主要用于定义排序规则，其中只有一个 compareTo()方法；
- 2) java.util.Comparator 是挽救的比较器操作，需要设置单独的比较器规则类实现排序，其中有 compare()方法。

## 28 类库使用案例

### ✧ 28.1 StringBuffer 使用

StringBuffer 对象 append()方法添加 26 个小写字母，逆序，删除前 5 个字符。

```
private static String test() {
    StringBuffer sb = new StringBuffer();
    for (char c = 'a'; c <= 'z'; c++) {
        sb.append(c);
    }
    sb.reverse();
    sb.delete(0, 5);
    return sb.toString();
}

public static void main(String[] args) {
    System.out.println(test()); // utsrqponmlkjihgfedcba
}
```

### ✧ 28.2 随机数组

使用 Random 类产生 5 个[10, 100)的随机整数的数组

```
private static Random random = new Random();
private static int[] getArray(int len) {
    len = len > 0 ? len : 5; // 小于0默认5个
    int[] arr = new int[len];
    for (int i = 0; i < len; i++) {
        int n = random.nextInt(100);
        // 10~99随机数
        if (n >= 10) {
            arr[i++] = n;
        }
    }
    return arr;
}

public static void main(String[] args) {
    System.out.println(Arrays.toString(getArray(5)));
}
```

### ✧ 28.3 验证 IP 地址

编写正则表达式，判断给定 IP 地址是否合法。(0.0.0.0~255.255.255.255)

|          |          |
|----------|----------|
| 0~9:     | \d       |
| 10~99:   | [1-9]\d  |
| 100~199: | 1\d\d    |
| 200~249: | 2[0-4]\d |
| 250~255: | 25[0-5]  |

综上 0~255 可以写成: [1-9]? \d | 1 \d \d | 2 [0-4] \d | 25 [0-5]

```
public class Validator {
    private Validator() {}
}
```



```

    public static boolean validateIP(String ip) {
        if (ip == null || "".equals(ip)) {
            return false;
        }
        String re = "([1-9]?\\d|1\\d\\d|2[0-4]\\d|25[0-5])\\.([1-9]?\\d|1\\d\\d|2[0-4]\\d|25[0-5])";
        return ip.matches(re);
    }
}

String[] ips = new String[] { "192.168.1.1", "123.04.2.4", "12.34.5.67",
"9.8.7.321" };
for (String ip : ips) {
    System.out.println(ip + ":" + Validator.validateIP(ip));
}

```

结果:

```

192.168.1.1:true
123.04.2.4:false
12.34.5.67:true
9.8.7.321:false

```

#### ✧ 28.4 学生信息比较

按照"name:age:score|name:age|score|..."的格式定义字符串，如：

"maki:15:99|rin:15:55|nozomi:17:78|umi:16:90|kotori:16:88|nico:17:55"

要求将每组值分别保存到 Student 对象中，并对这些对象按照成绩从高到低进行排序，如果成绩相同，按照年龄从低到高排序。

```

class Student implements Comparable<Student> {
    private String name;
    private int age;
    private int score;
    public Student(String name, int age, int score) {
        this.name = name;
        this.age = age;
        this.score = score;
    }
    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + ", score=" + score +
        "]\n";
    }
    @Override
    public int compareTo(Student o) {
        if (this.score > o.score) {
            return -1;
        } else if (this.score < o.score) {

```

```

        return 1;
    } else {
        return this.age - o.age;
    }
}
}

public static Student[] parse(String s) {
    String[] ret = s.split("\\|");
    Student[] arr = new Student[ret.length];
    for (int i = 0; i < arr.length; i++) {
        String[] tmp = ret[i].split(":");
        arr[i] = new Student(tmp[0], Integer.parseInt(tmp[1]),
Integer.parseInt(tmp[2]));
    }
    Arrays.sort(arr);
    return arr;
}

String s =
"maki:15:99|rin:15:55|nozomi:17:78|umi:16:90|kotori:16:88|nico:17:55";
System.out.println(Arrays.toString(parse(s)));

```

结果:

```

[
  Student [name=maki, age=15, score=99],
  Student [name=umi, age=16, score=90],
  Student [name=kotori, age=16, score=88],
  Student [name=nozomi, age=17, score=78],
  Student [name=rin, age=15, score=55],
  Student [name=nico, age=17, score=55]
]

```