

复旦大学计算机科学技术学院



编程方法与技术

9.1. 上次课复习

周扬帆

2021-2022第一学期

Java this关键字

```
ClassA objectA = new ClassA();
```

```
...
```

```
class ClassA {  
    public ClassA() {  
        this(1);  
        ...  
    }  
    public ClassA(int i) {  
        ...  
    }  
}
```

调用另一个构造方法

1. 第一行
2. 只能调用一次

- 构造方法里调用另一个构造方法
- 为什么有这个调用另一个构造方法的需要

Java this关键字

```
ClassA objectA = new ClassA();
```

```
...
```

```
class ClassA {  
    public ClassA init () {  
        ...  
        return this;  
    }  
    public ClassA connect () {  
        ...  
        return this;  
    }  
    public ClassA disconnect () {  
        ...  
        return this;  
    }  
    public ClassA setTimeout(int) { ...}  
    public ClassA setLogLevel(int) { ...}  
}
```

```
} ■ 实现Fluent Interface
```

```
■ ...
```

```
objectA.init()  
    .connect()  
    .disconnect();
```

```
objectA.init()  
    .setTimeout(1)  
    .connect()  
    .disconnect();
```

```
objectA.init()  
    .setTimeout(1)  
    .setLogLevel(0)  
    .connect()  
    .disconnect();
```

```
objectA.init()  
    .setLogLevel(0)  
    .setTimeout(1)  
    .connect()  
    .disconnect();
```

JavaScript函数的this

- this引用的指向和函数的调用方式有关
- 函数调用方式
 - 普通的函数调用
 - 对象的方法调用
 - obj.b()和 `var c=obj.b; c();` 的区别
 - 构造函数调用
 - `var obj = new func();`
 - Apply/Call调用
 - `func.apply(obj, args)`
 - `func.call(obj, arg1, arg2, ...)`

JavaScript函数的this

```
function Test () {  
    var b = function () {  
        this.c = 10;  
    }  
    this.test = function () {  
        b();  
    }  
}  
var test = new Test ();  
test.test();  
console.log(test.c);  
console.log(c);
```

undefined
10

Java File类

□ File类

- `import java.io.File;`
- 用来与操作系统交互，实现各种文件操作
 - 删除、重命名、创建文件夹等
 - 判断文件是否存在，文件属性，是否是目录等
 - 获得文件大小、路径、名字等

□ 构造方法

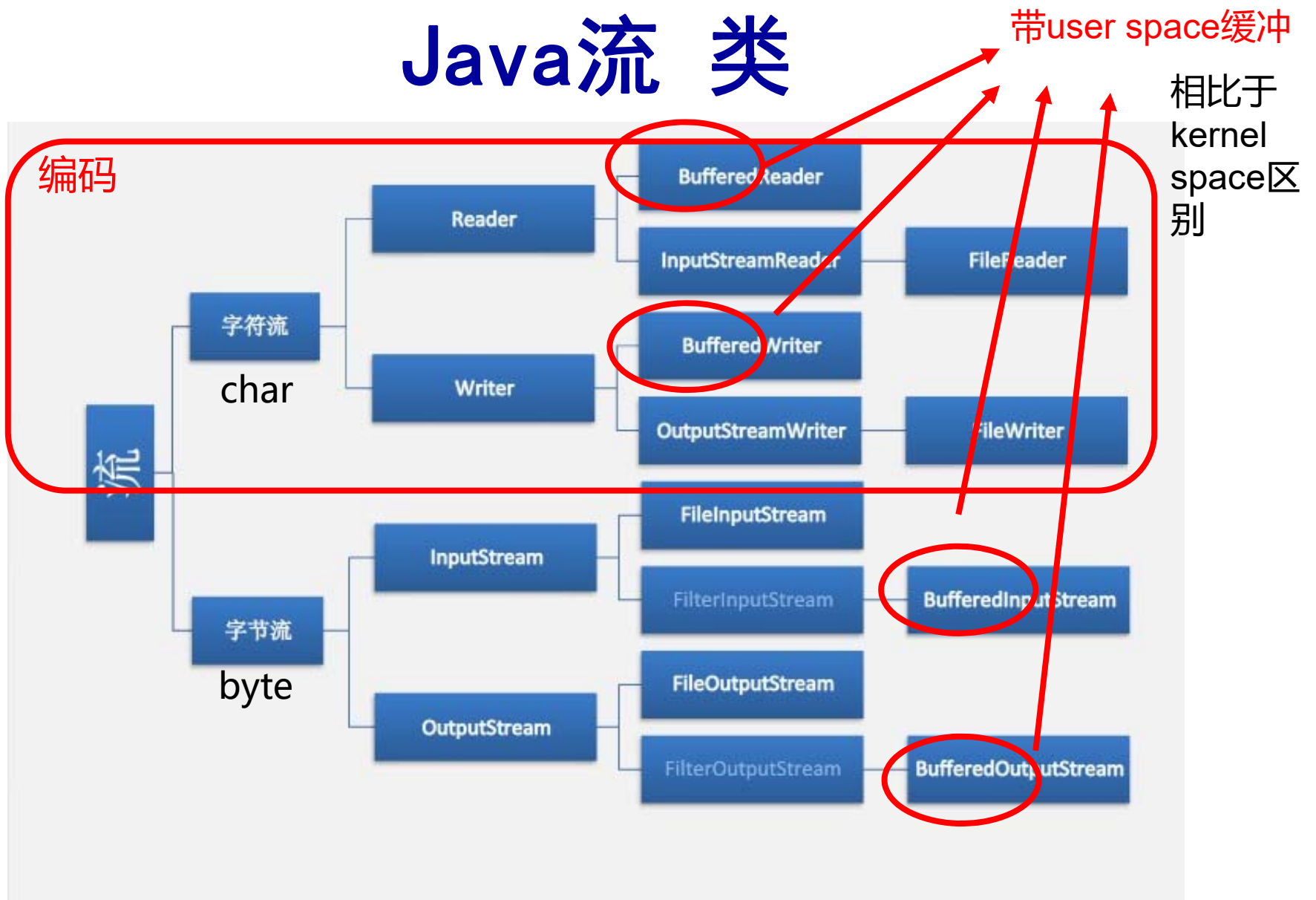
- `File file = new File(String pathName);`

另：

File.separator : Linux的 / 和Windows的 \

File.pathSeparator: Linux的: 和windows的 ;

Java流 类



文件读写 - 编码

❑ 错误的编码读入，同样错误的编码写回

```
InputStreamReader isr= new InputStreamReader(new FileInputStream(srcFile), "gb2312");
OutputStreamWriter osw= new OutputStreamWriter (new FileOutputStream(destFile), "gb2312");
char[] bytes = new char[16];
int size = 0;
while ((size = isr.read(bytes)) >= 0) {
    osw.write(bytes, 0, size);
}
isr.close();
osw.close();
```

- 文件srcFile的原编码是UTF-8
- destFile和srcFile会完全一样吗
- 怎么判断编码

This is a test
这是一个测试
這是一個測試
詰

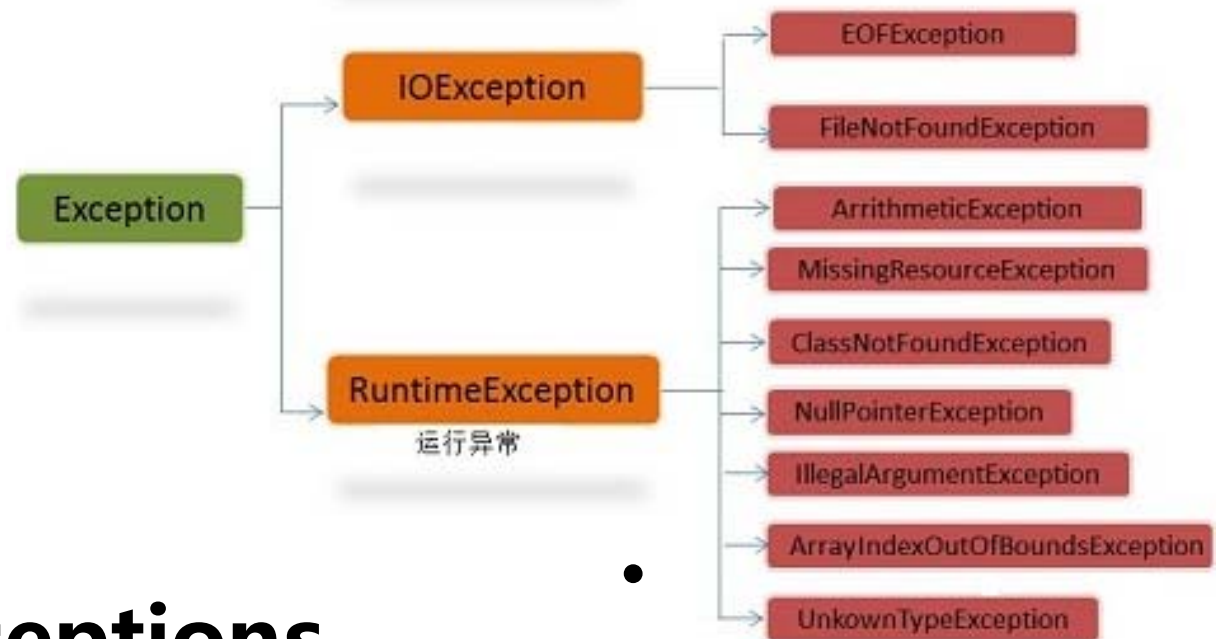


This is a test
这是一个测试
?是一??
?

Java异常

- **程序运行时会遇到很多异常**
 - 文件找不到
 - 读写文件时发生IO错误
 - 数组越界
- **在运行时通过检查**
 - 捕捉异常并处理、防止程序崩溃

Java异常



❑ Checked exceptions

- 提供机制，强制程序员写异常处理
- 什么是需要强制的？

❑ Unchecked exceptions

- 提供机制，让程序员可以在发生异常后，进行处理

程序员

环境

Java异常的捕捉

□ try/catch/finally

□ 用法

```
try {  
    //可能会抛出异常的语句  
}  
catch (XException e) {  
    //异常处理的语句  
}  
catch (YException e) {  
    //异常处理的语句  
}  
finally {  
    //最后需要执行的语句  
}
```

Java异常的捕捉

```
try {  
    //可能会抛出异常的语句  
}  
catch (XXException e) {  
    //异常处理的语句  
}  
catch (XXException e) {  
    //异常处理的语句  
}  
finally {  
    //最后需要执行的语句  
}
```

遇到异常，按顺序查下来

多态特性：

**如果捕捉到子类的异常
会进入父类的catch**

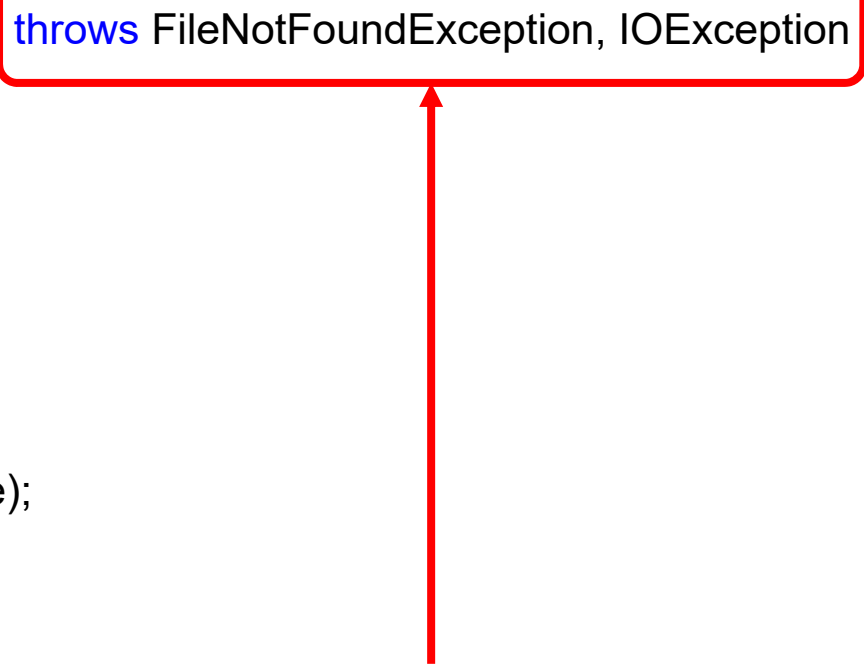
**因此父类的catch永远不能
在子类的catch前面，否则报错**

永远会执行！不管有没有异常

Java异常的抛出

❑ 方法可以不处理异常，而将异常抛出给调用者

```
public static void copy(String sFile, String dFile) throws FileNotFoundException, IOException {  
    File srcFile = new File(sFile);  
    File destFile = new File(dFile);  
    FileInputStream fin = null;  
    FileOutputStream fout = null;  
    fin = new FileInputStream(srcFile);  
    if (!destFile.exists()) {  
        destFile.createNewFile();  
    }  
    fout = new FileOutputStream(destFile);  
    byte[] bytes = new byte[1024];  
    while (fin.read(bytes) != -1) {  
        fout.write(bytes);  
        fout.flush();  
    }  
}
```



throws Exception

替换一下，然后在外面调用的时候
catch IOException还能catch得到吗？

Java自定义异常

- ❑ 异常也是一种Java类
- ❑ 可自定义自己的异常类

```
public class MyException extends Exception {  
    String message;  
    public MyException(String exceptionMessage) {  
        message = exceptionMessage;  
    }  
    public String getMessage() {  
        return message;  
    }  
}  
  
if(...) {  
    throw new MyException("Error Message");  
}
```

JavaScript异常捕捉

□ 一样的语法try/catch/finally

```
try {  
    var value;  
    console.log(value1);  
}  
catch (err) {  
    var txt = "Error description: " + err.message;  
    console.log(txt);  
}  
finally {  
    console.log('OK');  
}
```

先try, 有异常就catch, 不管有没有异常, 都finally

JavaScript异常throw

□ 自定义抛出异常信息

```
try {  
    console.log('Step 1');  
    throw 'My Exception';  
    console.log('Step 2');  
}  
catch (err) {  
    console.log(err);  
}
```

```
try {  
    console.log('Step 1');  
    throw new Error('My Exception');  
    console.log('Step 2');  
}  
catch (err) {  
    console.log(err);  
    //console.log(err.getMessage);  
}
```


复旦大学计算机科学技术学院



编程方法与技术

9.2. 日期

周扬帆

2021-2022第一学期

日期：Date类

```
import java.util.Date;  
import java.text.SimpleDateFormat;  
...  
Date date = new Date();
```

获取当前时间

```
long time = date.getTime();
```

获取从GMT 1970-01-01 00:00
至今的毫秒数

设置时间格式↓

```
SimpleDateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
```

```
String sysTime = df.format(date);
```

格式化时间

日期: Calendar 类

```
import java.util.Calendar;
```

```
...
```

```
Calendar c = Calendar.getInstance();
```

获取当前时间

```
int year = c.get(Calendar.YEAR);  
int month = c.get(Calendar.MONTH);  
int date = c.get(Calendar.DATE);  
int hour = c.get(Calendar.HOUR_OF_DAY);  
int minute = c.get(Calendar.MINUTE);  
int second = c.get(Calendar.SECOND);
```

获取各个属性的值

cal.get(Calendar.HOUR)
cal.get(Calendar.AM_PM)

```
c.set(Calendar.YEAR, 2017);  
c.set(Calendar.MONTH, 2);  
c.set(Calendar.DATE, 14);  
c.set(Calendar.HOUR_OF_DAY, 0);  
c.set(Calendar.MINUTE, 0);  
c.set(Calendar.SECOND, 0);
```

设置各个属性的值

日期: Calendar 类

```
import java.util.Calendar;  
import java.util.Date;  
import java.text.SimpleDateFormat;
```

```
...
```

```
Calendar c = Calendar.getInstance();
```

← 获取当前时间

```
Date time = c.getTime();
```

← Calendar转换为Date对象

```
c.setTime(time);
```

← Date转换为Calendar对象

```
String str = "2017-04-26";  
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
Date date = sdf.parse(str);
```

← 字符串格式的时间转换为Date对象

无聊冷知识 - 坑

```
try {  
    String str = "2021-15-02";  
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
    Date date = sdf.parse(str);  
    sdf = new SimpleDateFormat("yyyy-MM-dd");  
    String time = sdf.format(date);  
    System.out.println("Time: " + time);  
  
} catch (ParseException e) {  
    System.out.println("Format error");  
}
```

Time: 2022-03-02

```
Calendar c = Calendar.getInstance();  
int year = c.get(Calendar.YEAR);  
int month = c.get(Calendar.MONTH);  
System.out.println("Year: " + year + " Month: " + month);
```

Year: 2021 Month: 10(不是11)

无聊冷知识 - 坑

```
public static void main(String[] args) {  
    SimpleDateFormat yMd = new SimpleDateFormat("yyyy-MM-dd");  
    Calendar calendar = Calendar.getInstance();  
    calendar.set(Calendar.YEAR, 2021);  
    calendar.set(Calendar.MONTH, 11);  
    calendar.set(Calendar.DAY_OF_MONTH, 29);  
    System.out.println("yyyy-MM-dd = " + yMd.format(calendar.getTime()));  
  
    SimpleDateFormat YMd = new SimpleDateFormat("YYYY-MM-dd");  
    Calendar calendar2 = Calendar.getInstance();  
    calendar2.set(Calendar.YEAR, 2021);  
    calendar2.set(Calendar.MONTH, 11);  
    calendar2.set(Calendar.DAY_OF_MONTH, 29);  
    System.out.println("YYYY-MM-dd = " + YMd.format(calendar2.getTime()));  
}
```

2021-12-29

2022-12-29

复旦大学计算机科学技术学院



编程方法与技术

9.3. 定时器

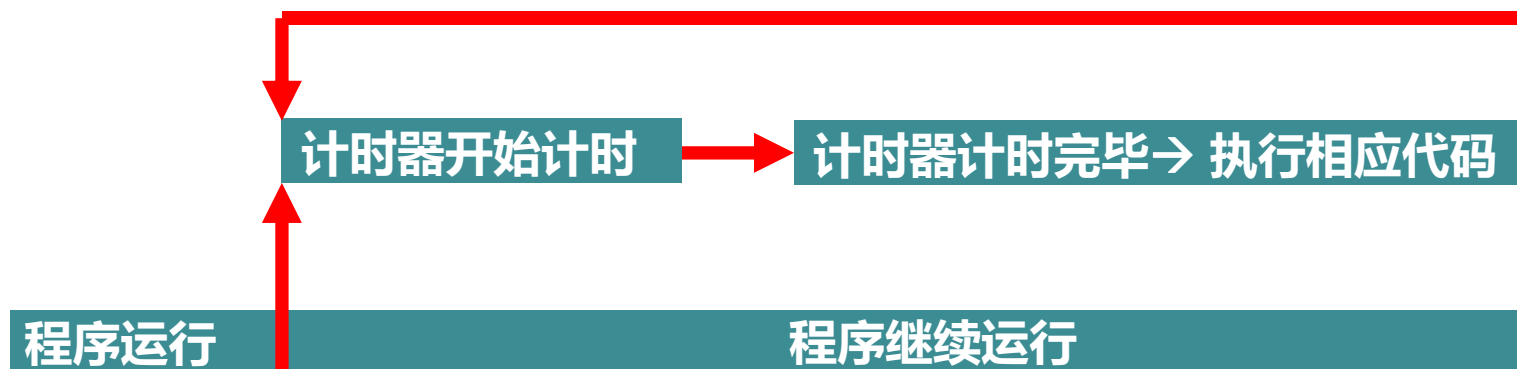
周扬帆

2021-2022第一学期

定时器：Timer

□ 使用场景：程序的周期性工作

- 每一秒钟刷新界面的一条文本信息
- 每一分钟重连网络资源
- 每一小时检查电子邮件
- 每一天检查版本更新



定时器实现方法例子

```
import java.util.Timer;
import java.util.TimerTask;
...
Timer timer = new Timer();
timer.schedule(new TimerTask() {
    public void run() {
        System.out.println("timer timeout: start run().");
        ...
    }
}, 2000); // 设定指定的时间time,此处为2000毫秒
```

- ❑ 使用了Timer类
- ❑ 使用Timer类的schedule方法设置定时器
 - 参数：一个TimerTask对象
 - 其他参数：控制启动时间，周期性等

TimerTask

```
class MyTask extends TimerTask {  
    public void run() {  
        System.out.println("timer timeout: start run().");  
        ...  
    }  
}
```

- ❑ **TimerTask: 定义定时任务**
- ❑ **程序需继承TimerTask抽象类**
 - 实现其**run方法**
 - **run(): 实现定时器timeout之后的操作**

Timer的schedule方法

`void schedule(TimerTask task, long delay)`

安排在指定延迟后执行task

`void schedule(TimerTask task, long delay, long period)`

安排task在指定的延迟后开始, 并按指定的周期重复执行

`void schedule(TimerTask task, Date time)`

安排在指定的时间执行task

`void schedule(TimerTask task, Date firstTime, long period)`

安排task在指定的时间开始, 并按指定的周期重复执行

`void scheduleAtFixedRate(TimerTask task, long delay, long period)`

安排task在指定的延迟后开始, 并按指定的周期重复执行

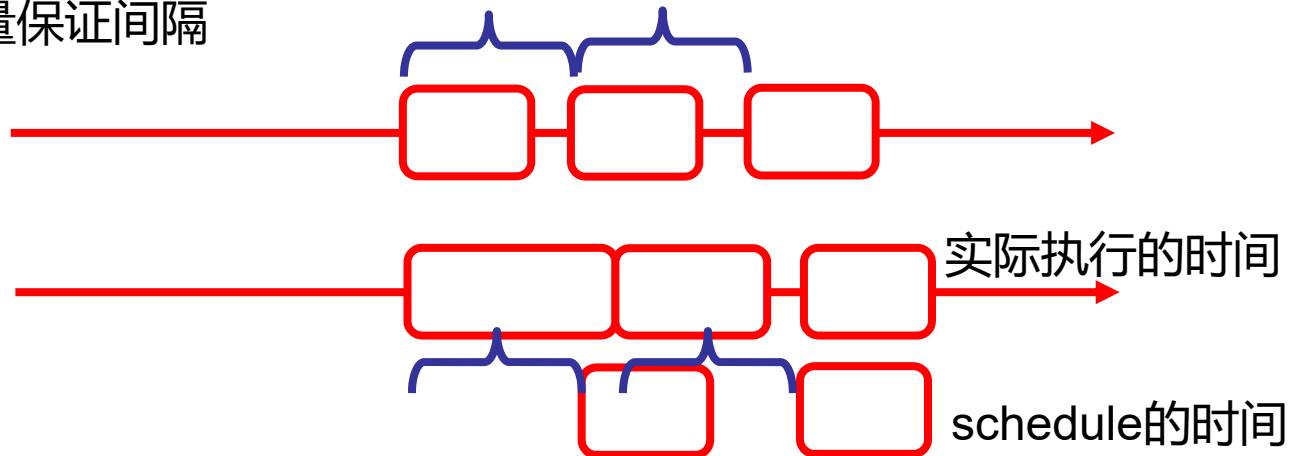
`void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)`

安排task在指定的时间开始, 并按指定的周期重复执行

Timer的schedule方法

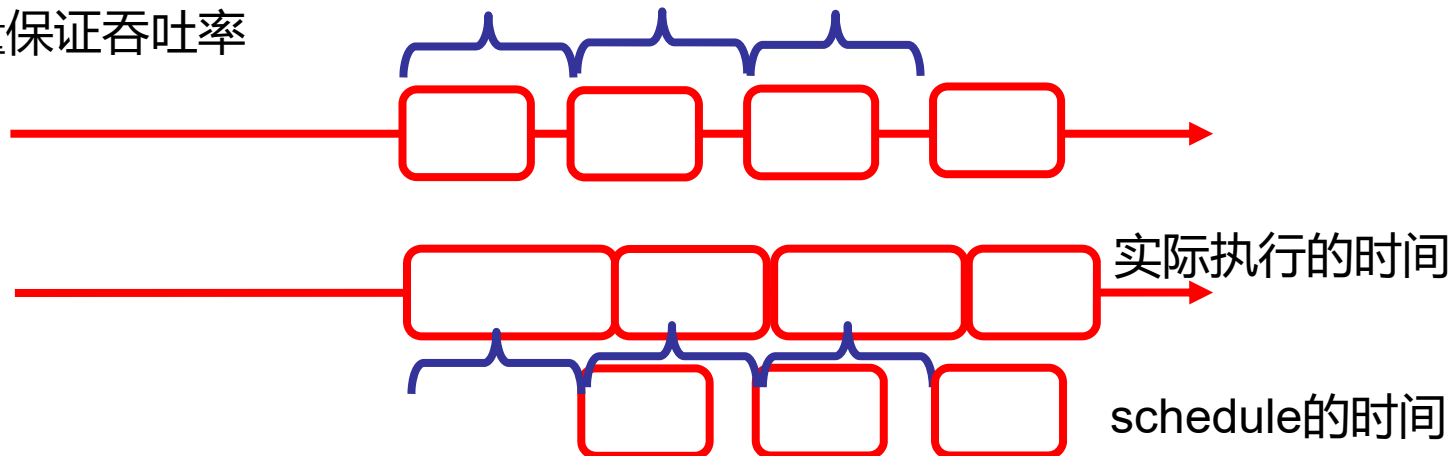
`void schedule(TimerTask task, long delay, long period)`

尽量保证间隔



`void scheduleAtFixedRate(TimerTask task, long delay, long period)`

尽量保证吞吐率



Timer的线程

`Thread.currentThread().getId()`

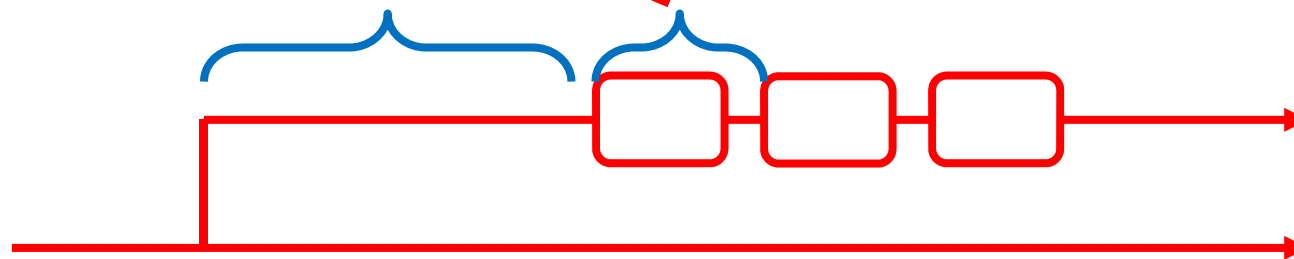
```
import java.util.Timer;
import java.util.TimerTask;
...
Timer timer = new Timer();
timer.schedule( new TimerTask() {
    public void run() {
        System.out.println("timer timeout: start run().");
        ...
    }
}, 2000); // 设定指定的时间time,此处为2000毫秒
...
```

Timer的取消

```
import java.util.Timer;
import java.util.TimerTask;
...
Timer timer = new Timer();
timer.schedule( new TimerTask() {
    public void run() {
        System.out.println("timer timeout: start run().");
        ...
    }
}, 2000); // 设定指定的时间time,此处为2000毫秒
...
timer.cancel();
```

Timer的取消

```
import java.util.Timer;
import java.util.TimerTask;
...
Timer timer = new Timer();
timer.schedule( new TimerTask() {
    public void run() {
        System.out.println("timer timeout: start run().");
        ...
    }
}, 2000, 1000); // 设定指定的时间time,此处为2000毫秒
...
timer.cancel();
```



复旦大学计算机科学技术学院



编程方法与技术

9.4. 并发与多线程：入门

周扬帆

2021-2022第一学期

线程：Thread

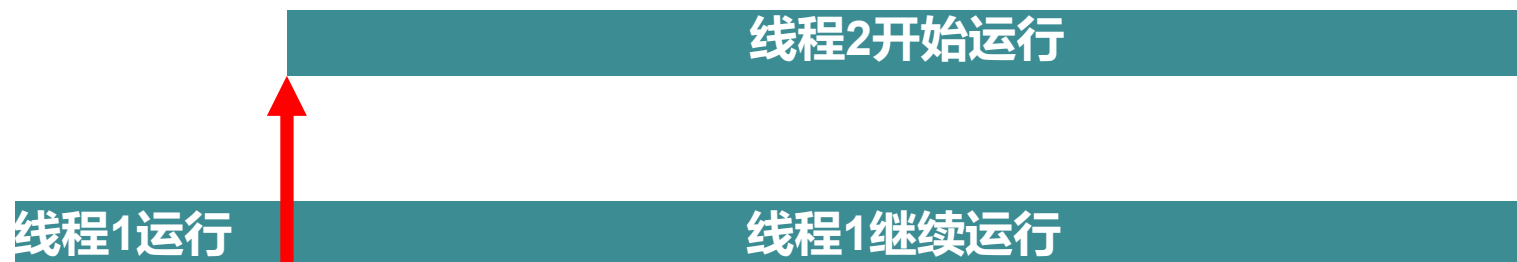
- ❑ 使用场景：程序的多任务执行
- ❑ 对于一个CPU核，计算是sequential的，多线程还有什么好处？



线程：Thread

□ 使用场景：程序的多任务执行

- 后台下载文件
- 后台更新数据库
- 后台监控、日志记录
- 服务器端并发用户请求处理
- ...



单线程的任务排队

□ 考虑四项不相关的任务

- 打印机、磁盘、数据库和显示屏
- 不相关，但任务也需要排队执行

```
class MyClass {  
    public static void main(String args[]) {  
        print_a_file();  
        manipulate_another_file();  
        access_database();  
        draw_picture_on_screen();  
    }  
}
```

- CPU运算很快，而I/O往往很慢
- 让CPU等待I/O效率太低
- 挂起，等待I/O就绪

解决排队

□ 方案1 – 多个进程 (process)

- 每项任务一个进程
- 阻塞时另一个程序可以运行
- 代价：设置进程要占用处理器和内存、通讯不方便

□ 方案2 – 多个线程 (thread)

- 创建过程相对轻，也被称为轻型进程 (LWP)
- 多个线程活动于同个进程的作用域，允许协作和数据交换

线程 vs. 进程

□ 进程：资源分配的基本单位

- 独立运行的程序
- 有自己的内存地址空间

□ 线程：CPU调度的基本单位

- 包含在一个进程中的一个串行控制流 (control flow)
- 一个进程可以有多个同时运行的线程
- 线程创建、销毁的**开销**比进程低
- 线程间**通讯**比进程间通讯更容易实现
- 内存空间共享

Java的线程

- **每个 Java 程序都使用线程**
 - 程序主线程随程序启动，调用main方法
 - 其他线程：垃圾收集、图形界面等等
- **作用例子：响应更快的 UI**
 - 较长的任务放到事件处理线程，UI主线程在任务的执行过程中就可以继续处理UI 事件
 - 防止UI被Freeze
- **利用多核多处理器系统**
- ...

多线程编程方法1

□ Thread类的定义

- Thread是抽象类，具体实现，需重载run方法

```
Class MyThread extends Thread{  
    @Override  
    public void run(){  
        System.out.println("Hello World!");  
    }  
}
```

□ 线程的启动：Thread对象的调用

- 创建实例并调用start()方法

```
public static void main(String[] args) {  
    MyThread myThread = new MyThread();  
    myThread.start();  
}
```

“填空”式编程

多线程编程方法2

□ 使用Runnable接口来实现多线程

```
class MyRunnable implements Runnable{  
    @Override  
    public void run() {  
        System.out.println("Hello World!");  
    }  
}  
  
public static void main(String[] args) {  
    Thread myThread = new Thread(new MyRunnable());  
    myThread.start();  
}
```

- 方法1直接继承Thread类限制了继承关系
 - 不能多重继承
- 方法2使用接口更为灵活
 - Thread构造时，传入Runnable对象
 - Runnable接口：实现run方法

Java的线程

□ 简单，但有时有风险

- 多个线程访问同一数据项，需要协调 → 线程安全性

□ 不要做过头

- 消耗一定资源：多了会降低整体性能
- 在单处理器系统中，多线程不会使主要消耗 CPU 的程序更快
- 多线程实现多个I/O intensive的任务呢？

线程安全性

□ 两个线程如果访问同一个数据?

```
class MyRunnable implements Runnable{  
    static Counter counter = new Counter();  
    ...  
    public void run() {  
        counter.changeCount(..);  
    }  
}
```

对对象counter进行操作

```
...  
(new Thread(new MyRunnable2())).start();  
(new Thread(new MyRunnable2())).start();
```



synchronized关键字

□ 可修饰方法

```
public synchronized void myMethod(... ) {  
    ...  
}
```

- 给这个方法加上一个锁(lock)
- 同时，只有一个线程可以在执行这个方法
 - 谁执行，谁获得锁
 - return释放锁
- 请测试一下，如果一个类两个方法都用synchronized修饰，能不能同时进入

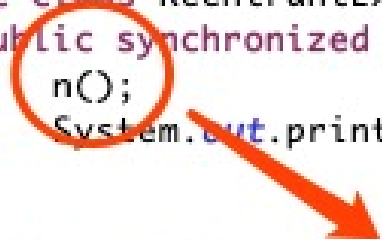
□ 可修饰块(block)

```
synchronized (对象的引用) {  
    // 执行时，该共享对象被锁  
    // 只能同时被一个线程执行  
}
```

synchronized关键字

□ 重入

```
public class ReentrantExample {  
    public synchronized void m() {  
        n();  
        System.out.println("this is m() method");  
    }  
  
    public synchronized void n() {  
        System.out.println("this is n() method");  
    }  
  
    public static void main(String args[]) {  
        final ReentrantExample re = new ReentrantExample();  
  
        Thread t1 = new Thread() {  
            public void run() {  
                re.m(); // calling method of Reentrant class  
            }  
        };  
        t1.start();  
    }  
}
```

A red circle highlights the 'n()' call inside the 'm()' method. A red arrow points from this circle to the 'n()' method definition below, illustrating that 'm()' is calling 'n()' on the same object, which is allowed for synchronized methods (reentrancy).

synchronized关键字

□ 可修饰方法

```
public synchronized void myMethod(... ) {  
    ...  
}
```

- 请测试一下，如果一个类两个方法都用synchronized修饰，能不能同时进入
 - 不一定？
 - 普通方法/静态方法
 - 猜原因？

□ synchronized编程猪头了的风险？

- 死锁
- 长时间等待，并行度不高

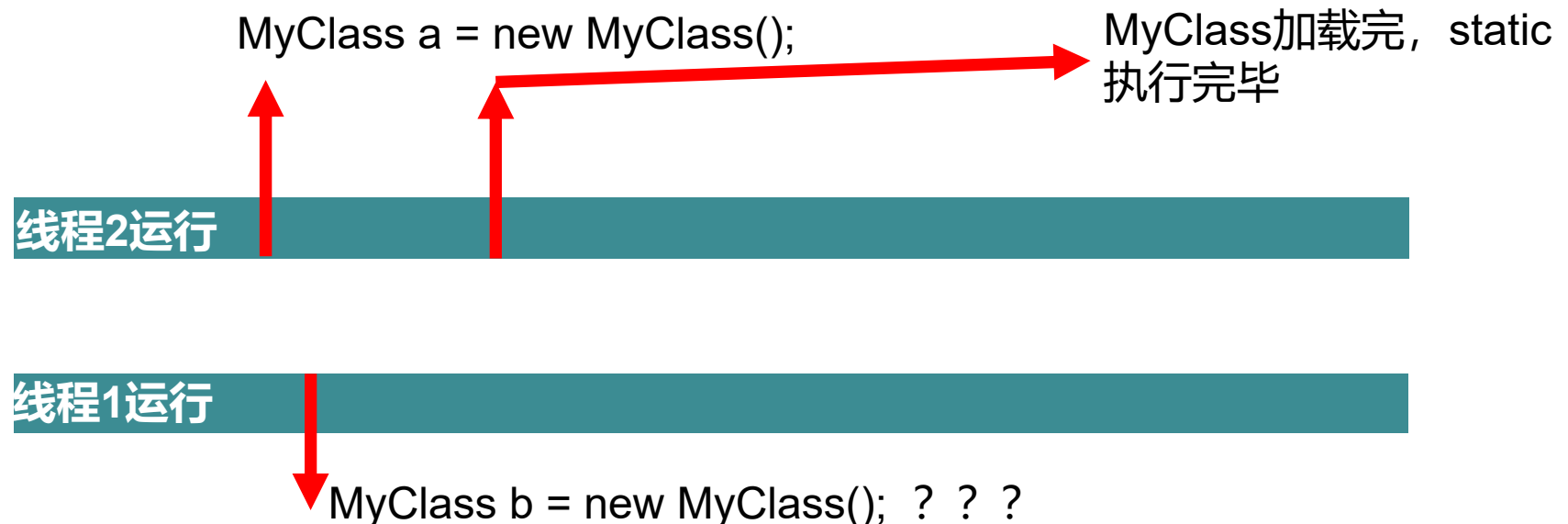
Java的线程

□ 关于类的初始化，那些static的东西

■ 多线程下，执行一次？

→ 请测试一下多线程下static块执行顺序

■ If so, 如何做到？



思考

- 理解线程与进程的关系
- 思考到目前为止学到的实现多线程的方式
- 理解线程不安全的情境
- 了解线程调度和时间分片、上下文切换(context switch)
- 了解进程、线程之间的通信方式
 - 这里说的通信，本质是什么？

复旦大学计算机科学技术学院



编程方法与技术

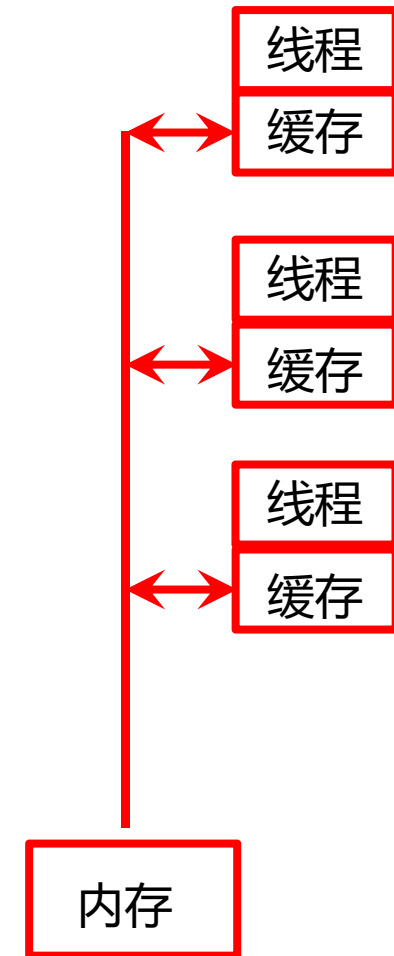
9.5.并发与多线程：一致性和原子性

周扬帆

2021-2022第一学期

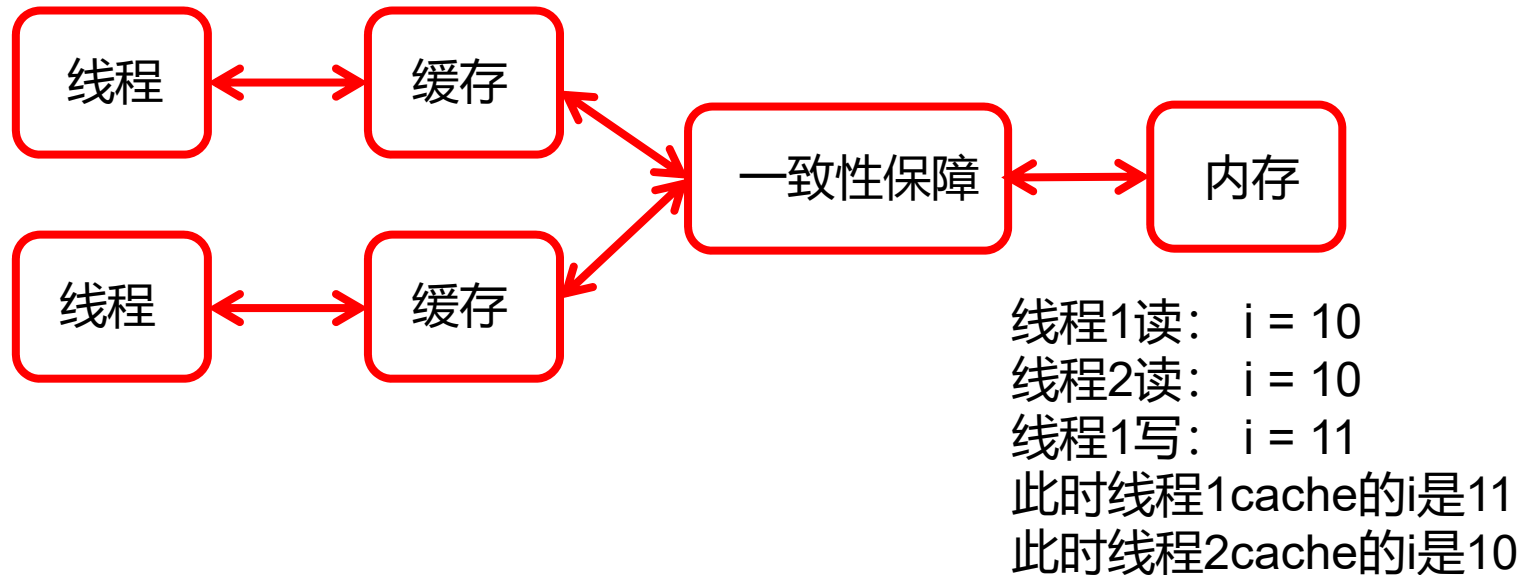
内存模型

- ❑ 指令：控制流-CPU的指令执行
- ❑ 数据：内存
 - 内存读写太慢（相比于CPU）
 - 引入高速缓存(cache)
 - 数据从内存拷贝到缓存
 - CPU访问缓存数据
 - 缓存数据写回内存
- ❑ `i ++`；在多线程方式下？



内存模型

□ 缓存一致性



Java的一致性和原子性保障

□ **volatile**关键字：修饰变量

- 每次遇到volatile变量，会强制缓存更新

```
volatile int x;
```

```
...
```

```
x = x + 1;
```

- 因此volatile变量的值与此时的内存是一致的
- 本质上，volatile加入了**内存屏障**
 - volatile 写操作前插入 **StoreStore** 屏障，在写操作后插入 **StoreLoad** 屏障：前面要写的都写完，后面要读的还不能读
 - volatile 读操作前插入 **LoadLoad** 屏障，在读操作后插入 **LoadStore**屏障：前面要读的都读完，后面要写的还不能写

□ 有了volatile能不用锁保持同步吗？

- 等一下来看例子

内存模型

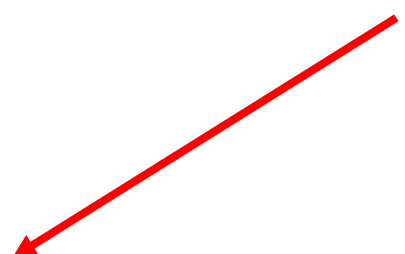
□ 操作原子性

- $i = 9$

- 可能分为两次操作，低16赋值，高16位赋值
- Java保证原子性(atomic): 操作的中间不会插入其他操作
- 否则 $i = -2$ 和 $i = 9$ 多线程下，可能 i 不等于-2也不等于9

一致性与原子操作示例

```
public class SyncDemo {  
    public volatile static int count = 0;  
    public static void inc() {  
        try {  
            Thread.sleep(1);  
        } catch (InterruptedException e) {  
        }  
        count++;  
    }  
    public static void main(String[] args) {  
        for (int i = 0; i < 5000; i++) {  
            new Thread(new Runnable() {  
                public void run() {  
                    SyncDemo.inc();  
                }  
            }).start();  
        }  
        // ... try {Thread.sleep(5000); } catch (InterruptedException e) { ...}  
        System.out.println("Counter.count=" + SyncDemo.count);  
    }  
}
```

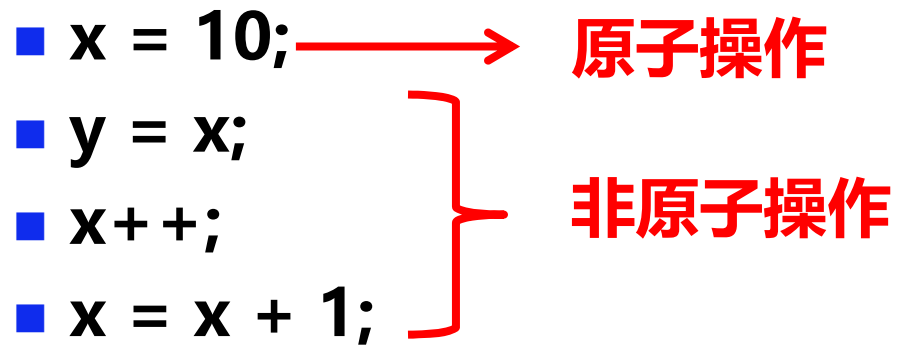


一致性与原子操作示例

```
public class TestAtomic {  
    private static volatile boolean bChanged;  
    public static void main(String[] args) throws InterruptedException {  
        new Thread() {  
            public void run() {  
                for(int i = 0; ;i++) {  
                    if(bChanged == !bChanged) {  
                        System.out.println("OK: i = " + i);  
                        System.exit(0);  
                    }  
                }  
            }  
        }.start();  
        Thread.sleep(1);  
        new Thread() {  
            public void run() {  
                for(;;) {  
                    bChanged = !bChanged;  
                }  
            }  
        }.start();  
    }  
}
```

Java的一致性和原子性保障

□ 原子操作

- `x = 10;` → 原子操作
 - `y = x;`
 - `x++;`
 - `x = x + 1;`
- 非原子操作
- 

Java的原子性与一致性

□ 设计时

- 考虑哪些操作需要保证原子性
- 考虑哪些数据需要保证线程间的一致
- 用锁机制配合状态变量来实现
 - synchronized块
 - wait/notify

思考

- ❑ 结合数据库与操作系统的相关知识，思考如何实现一致性和原子性，思考其副作用及如何降低
- ❑ 了解乐观访问控制(OCC)和悲观访问控制(PCC)
- ❑ 了解Java的阻塞队列
- ❑ 了解对象的monitor

复旦大学计算机科学技术学院



编程方法与技术

9.6. 单例模式

周扬帆

2021-2022第一学期

日志

□ 需求

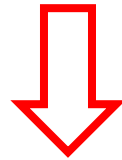
■ 实现日志功能

→ logError

→ logWarning

→ logInfo

```
System.out.println("[Warning]: " + "connection failed");
```



```
logger.logWarning("connection failed");
```

日志

□ 朴素的设计

■ 实现Logger类, 实现相应的方法

```
Logger logger = new Logger("log.txt");  
logger.logError("Conf file not found.");
```

■ 问题

→ Logger这种东西, 整个进程只要一个就够了, 不需要那么多对象

→ 省空间

单例模式

□ 例: instance

- 一个类的实例化的对象，叫这个类的instance

□ 单例模式

- 保证一个类，在程序中，只有一个**实例**

□ 作用

- **公共的**管理模块
 - 如公共数据管理
 - 日志记录
- **公共的**不宜用静态方法实现的功能
 - 初始化 → 访问: 静态方法 vs 单例
- 节约资源，方便同步，高效...

单例模式

```
class Log {  
    private static Log instance = null;  
    public static Log getInstance() {  
        if (instance == null) {  
            Log instance = new Log ();  
        }  
        return instance;  
    }  
    private Log () {  
        ... //初始化, 如打开日志文件  
    }  
    public void logError (String log) {  
        ...  
    }  
}
```

❑ 有bug!
■ 多线程下?

```
Log.getInstance().logError("This is a log record");
```

单例模式

```
class Log {  
    private static Log instance = null;  
    public static synchronized Log getInstance() {  
        if (instance == null) {  
            Log instance = new Log ();  
        }  
        return instance;  
    }  
    private Log () {  
        ... //初始化, 如打开日志文件  
    }  
    public void logError (String log) {  
        ...  
    }  
}
```

加锁

□ 效率问题?

```
Log.getInstance().logError("This is a log record");
```

单例模式

```
class Log {  
    private static Log instance = null;  
    public static Log getInstance() {  
        if (instance == null) {  
            synchronized (作为锁的对象) {  
                if (instance == null) {  
                    Log instance = new Log ();  
                }  
            }  
        }  
        return instance;  
    }  
    private Log () {  
        ... //初始化, 如打开日志文件  
    }  
    public void logError(String log) {  
        ...  
    }  
}  
Log.getInstance().logError("This is a log record");
```

Log.class

- 双重检验+锁
- 为什么效率高?

单例模式

```
class Log {  
    private static Log instance = new Log ();  
    public static Log getInstance() {  
        return instance;  
    }  
    private Log () {  
        ... //初始化, 如打开日志文件  
    }  
    public void logError (String log) {  
        ...  
    }  
}  
Log.getInstance().logError("This is a log record");
```

□ 类加载时初始化

- 问题?
- 如果初始化很慢 → 类加载很慢

单例模式

```
class Log {  
    private static class LogHolder {  
        private static final Log INSTANCE = new Log();  
    }  
    public static Log getInstance() {  
        return LogHolder.INSTANCE;  
    }  
    private Log () {  
        ... //初始化, 如打开日志文件  
    }  
    public void logError (String log) {  
        ...  
    }  
}  
Log.getInstance().logError("This is a log record");
```

- 不调用getInstance不会加载LogHolder
 - 不会new Log()

复旦大学计算机科学技术学院



编程方法与技术

9.7. 练习

周扬帆

2021-2022第一学期

课堂练习

- 实现100个线程
- 每个线程循环10次实现counter的自增
 - sleep 100毫秒
- 使用同步机制，使得
 - 最后counter = 1000
- 比较不使用同步机制，最后counter的值