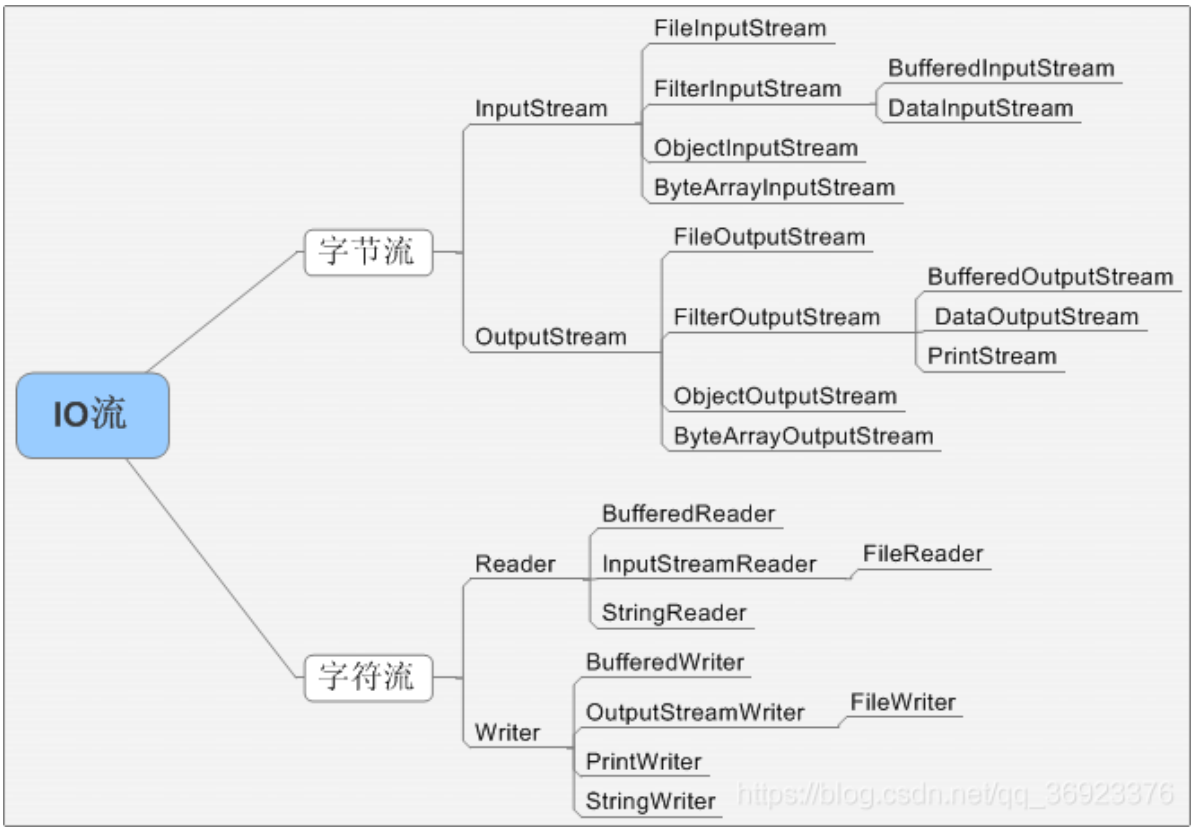
I/O



- 将字符串写入和写出

```
public class Main {
    public static void main(String[] args) {
        File file = new File("C:\\Users\\Administrator\\Desktop\\a.txt");
        FileOutputStream outputStream = null;
        FileInputStream inputStream = null;
        BufferedOutputStream bufferedOutputStream = null;
        try {
            outputStream = new FileOutputStream(file);
            bufferedOutputStream = new BufferedOutputStream(outputStream);
            inputStream = new FileInputStream(file);
            String str = "I love Java";
            byte[] bytes = str.getBytes(StandardCharsets.UTF_8);
            //将字符串写出到文件
            bufferedOutputStream.write(bytes);
            byte[] inByte = new byte[1024 * 1024 * 4];
            int len = 0;
            //将字符串写入到控制台
            while ((len = inputStream.read(inByte)) != -1) {
                System.out.println(new String(inByte, 0, len));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
//              关闭流，注意关闭顺序
            try {
                if (bufferedOutputStream != null) {
                    bufferedOutputStream.close();
```

```
                }
                if (inputStream != null) {
                    inputStream.close();
                }
                if (outputStream != null) {
                    outputStream.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- 将对象序列化到文件中

```
public class Main {
    public static void main(String[] args) {
        File file = new File("C:\\Users\\Administrator\\Desktop\\Person.xml");
        FileOutputStream outputStream = null;
        FileInputStream inputStream = null;
        ObjectOutputStream objectOutputStream = null;
        ObjectInputStream objectInputStream = null;
        try {
            outputStream = new FileOutputStream(file);
            objectOutputStream = new ObjectOutputStream(outputStream);
            //对象序列化到文件中
            objectOutputStream.writeObject(new Person(12, "Tom"));
            inputStream = new FileInputStream(file);
            objectInputStream = new ObjectInputStream(inputStream);
            //将对象序列化进来
            Person person = (Person) objectInputStream.readObject();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                //关闭资源,注意顺序
                if (objectInputStream != null) {
                    objectInputStream.close();
                }
                if (objectOutputStream != null) {
                    objectOutputStream.close();;
                }
                if (inputStream != null) {
                    inputStream.close();
                }
                if (outputStream != null) {
                    outputStream.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

BIO NIO AIO

Java BIO：`同步并阻塞`（传统阻塞型），服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不作任何事情会造成不必要的线程开销。

Java NIO：`同步非阻塞`，服务器实现模式为一个线程处理多个请求(连接)，即客户端发送的连接请求会被注册到多路复用器上，多路复用器轮询到有 I/O 请求就会进行处理。

Java AIO：`异步非阻塞`，AIO 引入了异步通道的概念，采用了 Proactor 模式，简化了程序编写，有效的请求才启动线程，它的特点是先由操作系统完成后才通知服务端程序启动线程去处理，一般适用于连接数较多且连接时间较长的应用。

- `同步阻塞`：你到饭馆点餐，然后在那等着，还要一边喊：好了没啊！
- `同步非阻塞`：在饭馆点完餐，就去遛狗了。不过溜一会儿，就回饭馆喊一声：好了没啊！
- `异步阻塞`：遛狗的时候，接到饭馆电话，说饭做好了，让您亲自去拿。
- `异步非阻塞`：饭馆打电话说，我们知道您的位置，一会给你送过来，安心遛狗就可以了。

使用场景：

- BIO 方式适用于`连接数比较小且固定`的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 之前唯一的选择，程序较为简单容易理解。
- NIO 方式适用于`连接数目多且连接比较短`的架构，比如聊天服务器，弹幕系统，服务器间通讯等，编程比较复杂，JDK1.4 开始支持。
- AIO 方式适用于`连接数目多且连接比较长`的架构，比如相册服务器，充分调用 OS 参与并发操作，变成比较复杂，JDK7 开始支持。

Reflecting

定义

JAVA反射机制是在运行状态中，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，都能够调用它的任意方法和属性；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

Java反射中首先是要获取需要反射类的字节码，获取字节码有三种方法：Class.forName(className) 类名.class 实例化的类 类.getClass(), 然后将字节码中的方法、变量、构造函数等映射成相应的Method、Field、Constructor等类，这些类提供了丰富的方法供我们使用。

- 先建立一个Person类：

```
public class Person {
    private String name;
    private Integer age;
    private String sex;

    public Person() {
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
```

```
    public void setAge(Integer age) {
        this.age = age;
    }
    public Integer getAge() {
        return age;
    }

    public void talk(String words) {
        System.out.println(words);
    }
}
```

- 用反射来操作Person类

```
public class Reflecting {
    public static void main(String[] args) {
        try {
            //三种获取类的影射对象的方法
            Class<?> a = Class.forName("test.Person");
            Class<Person> b = Person.class;
            Person person = new Person();
            Class<? extends Person> c = person.getClass();

            //获取Class对象中的方法
            Method[] methods = a.getDeclaredMethods();
            for (Method method : methods) {
                if ("setName".equals(method.getName())) {
                    method.invoke(person, "Tom");
                }
                if ("talk".equals(method.getName())) {
                    method.invoke(person, "Hello Java!");
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

结果会输出："Hello Java!"，实例化的person对象的成员变量name属性也设置成了"Tom"。

Multireading

一、多线程的实现

1. **继承Tread类**

```
public class MyTread extends Thread {
    @Override
    public void run() {
        System.out.println("我是子线程....");
    }
}
```

```
public class TestThread {
    public static void main(String[] args) {
        MyTread myTread = new MyTread();
        myTread.start();
        System.out.println("我是主线程");
    }
}
```

结果：

我是主线程
我是子线程....

2. **实现Runnable接口**

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("我是子线程");
    }
}
```

```
public class TestThread {
    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();
        System.out.println("我是主线程");
    }
}
```

二、线程的生命周期

5中状态：

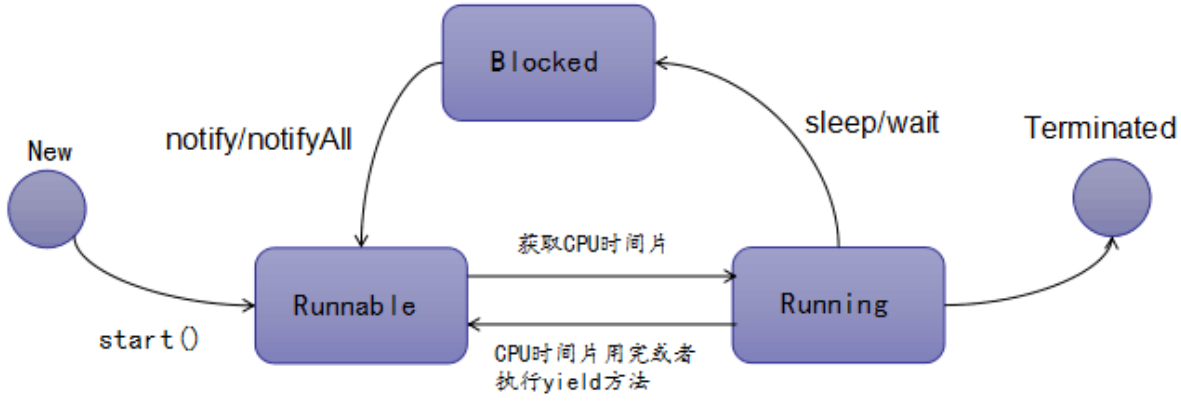
出生：new Thread()

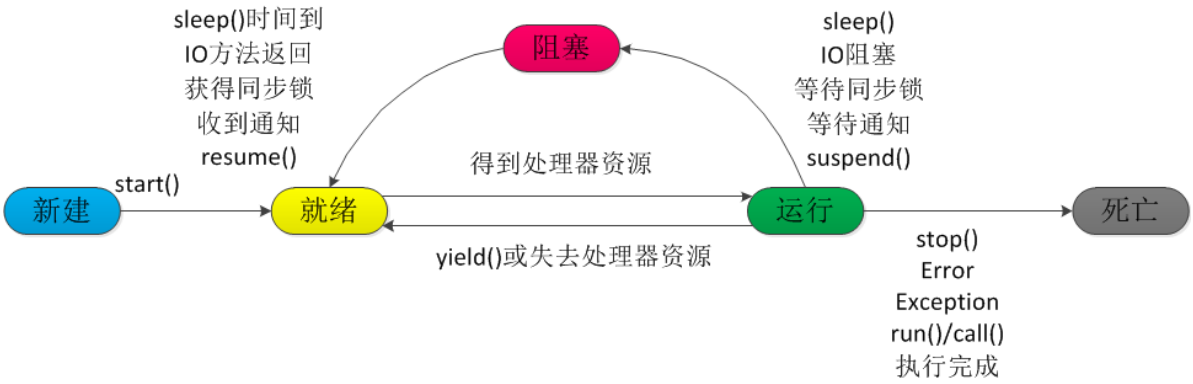
就绪: start()

运行

暂停：sleep()休眠、wait()等待

死亡：run()执行完、Error或Exception





三、线程的同步

1、同步块

```
public class SynchronizedTest implements Runnable {
    int ticket = 10;
    @Override
    public void run() {
        while (true) {//设置无线循环
            synchronized (this) {
                if (ticket > 0) {//判断当前票数是否大于0
                    try {
                        Thread.sleep(100);//使当前线程休眠100毫秒
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println(Thread.currentThread().getName() + "正在售
卖第" + ticket-- + "张票");
                }
            }
        }
    }
}
```

```
public class TestThread {
    public static void main(String[] args) {
        SynchronizedTest t = new SynchronizedTest();
        Thread thread1 = new Thread(t, "thread1");
        Thread thread2 = new Thread(t, "thread2");
        Thread thread3 = new Thread(t, "thread3");
        Thread thread4 = new Thread(t, "thread4");
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
    }
}
```

执行结果：

thread1正在售卖第10张票
thread4正在售卖第9张票
thread3正在售卖第8张票
thread3正在售卖第7张票
thread2正在售卖第6张票
thread2正在售卖第5张票
thread2正在售卖第4张票
thread2正在售卖第3张票
thread2正在售卖第2张票
thread2正在售卖第1张票

2、同步方法

```
public class SynchronizedTest implements Runnable {
    int ticket = 10;
    @Override
    public void run() {
        while (true) {//设置无线循环
            doit();
        }
    }
    public synchronized void doit() {
        if (ticket > 0) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + "正在售卖第" +
ticket-- + "张票");
        }
    }
}
```