Java的访问限定

权限访问限定	只能在本类中访问	只能在同一个包中访问	可以在继承关系下访问	可以任意访问
private	√	×	×	×
默认不写	√	√	×	×
protected	√	√	√	×
public	√	√	√	√

this关键字

this关键字，只能在方法的内部使用，表示的是调用这个方法的那个对象的**引用**，this关键字的使用和平时new出来的对象引用的使用没什么区别。

需要注意的是，如果是在方法内部调用同一个类的另一个方法，就不用使用this关键字了。直接调用方法就行，因为即使你不写，编译器会自动帮你加上this关键字，但是你写的话也不会出错。

this关键字的使用时机是，只有当有明确要求指出要得到调用当前方法的这个引用的时候，方使用this关键字。

```
public class demo04 {
    public static void main(String[] args) {
        new Person().eat(new Apple());
    }
}

/**
 * 人类：
 * 人有一个动作，吃，吃的时候传入一个苹果
 */
class Person {
    public void eat(Apple apple) {
        Apple peeled = apple.getPeeler();
        System.out.println("yummy");
    }
}

/**
 * 削皮机：传入一个苹果就可把这个苹果削皮
 */
class Peeler {
    static Apple peel(Apple apple) {
        return apple;
    }
}

/**
 * 苹果类：假如由于某种特别的原因，削皮这个操作不能再削皮机这个类中完成，
```

```
 * 只能在苹果类中调用削皮机，这个时候，可以通过this关键字，把自己的apple
 * 实例，传入到削皮机的方法中
 */
class Apple {
    Apple getPeeler() {
        return Peeler.peel(this);
    }
}
```

上面的代码演示了，this关键字将当前对象传递给其它方法。

对于this关键字的核心理解很简单，其实当使用一个方法的时候，我们如果想知道，到底是谁使用了这个方法，那么我们就可以用this关键字，把这个对象召唤出来。

```
public class demo5 {
    public static void main(String[] args) {
        Student liming = new Student("黎明");
        System.out.println(liming);
    }
}

class Student{
    private String name;
    public Student(String name) {
        this.name = name;
        System.out.println(this);
    }
}
```

上面的代码我们做了一个实验，创建了一个学生类，重写了他的构造器，我们在构造器中打印了this关键字，并在main方法中打印了通过new创建出来的实例对象，看看这个this关键字和通过new出来的实例对象的引用liming，他们是不是指向同一个实例。

```
test_startic.Student@4554617c
test_startic.Student@4554617c
```

通过控制台的打印，我们看出来，他们的地址实际上是一摸一样的。也就是说，this关键字，指向的东西，就是new出来的那个对象。他们两个是同一个东西两种不同存在形式。

基本类型的默认值

类型	默认值
boolean	**false**
char	**null**
byte	**0**
short	**0**
int	**0**
long	**0L**
float	**0.0f**
double	**0.0d**

return的作用：

第一：退出方法

第二：返回函数的结果值

如果方法是void的，那么不管在方法中的哪里遇到return都表示在这里退出方法，如果方法不是void，那么return表示在这里必须放回确定类型的方法结果。

Java编码与解码

java使用16位的双字节存储，但是在实际文件存储的过程中数据编码有各种各样的字符集，需要正确操作，否则就会出现乱码。

字符集	说明
US-ASCII	即英文的ASCII编码
IOS-8859-1	拉丁文、包含中文、日文等
UTF-8	边长的unicode字符（1-3个字节），国际通用

编码: 字符串-->字节

```
String msg ="性命生命使命a";
//编码: 字节数组
 //默认使用工程的字符集
byte[] datas = msg.getBytes();
```

如上代码就实现了字符串的编码，将其转换为了字节

在编码时候可以设置使用那种字符集进行编码

```
String msg = "你好世界";
byte[] bytes = msg.getBytes(StandardCharsets.UTF_8);
```

解码: 字节->字符串

```
String msg = new String(datas,0,datas.length,"utf8");
```

如上代码就实现了Java的解码

同样，解码的时候可以指定以那种字符集进行解码

乱码的原因：

1. 字节数不够：就是说原来编码完成之后比如字节的长度有12，解码的时候字节只有10了，就缺少了东西，无法复原
2. 字符集不统一：编码和解码的时候使用的不是同一种字符集就会完全产生乱码

获取当前的工程路径：

```
System.getProperty("user.dir");
```

Java的复用类

Java中一切都是对象，而对象源自于类型，也就是类。而将类做到复用是简化代码的一项重要措施。

在Java中将类复用有两种方法：

- 在一个新的类中包含已经有的类的对象。

```
public class Book {
    private String name;
    private StoryBook storyBook;
}
```

如上代码，在Book类中我们又组合进了一个新的StoryBook类型的对象，这种方式就是Java中的组合，它可以达到代码复用的功能

- 继承。它按照现有的类型来创建新的类型，不必改变现有类的形式，采用现有类的形式并在其中添加新的代码。

```
public class StoryBook extends Book{
    String name;
}
```

如上代码，StoryBook是新的类型，但是他是从原有的Book类型中扩展而来的。

我们都知道，一个类中的字段或着参数是基本类型的时候，会自动初始化为零，但是对象类型的引用数据类型会被初始化为null，这个时候，如果你想用这个引用类型的字段调用它其中的方法，那一定会报空指针异常的错误，因为这个引用没有指向任何一个实际的对象。

组合语法

组合语法及其简单，只需将对象的引用置于新的类中即可。

初始化引用类型字段的时机

那么我们如何对引用类型的字段进行正确的初始化？下面有几个初始化的时机需要牢记。

在定义对象的地方

```
public class Book {
    private String name;
    private StoryBook storyBook = new StoryBook();
}
```

如上代码，在定义的时候就要使用new 关键字为其分配内存空间以对其初始化。

在类的构造器中

```
public class Book {
    private String name;
    private StoryBook storyBook = null;
    Book(){
         this.storyBook = new StoryBook();
    }
}
```

如上代码，可以在类的构造器中进行初始化，保证这个类的对象在使用的时候它其中的所有字段都是可用的。

在正要使用这个类型的对象之前

这种方式称之为惰性初始化。也就是不到使用这个对象的时候不会为他分配空间，这样能做到一定程度的性能提升。

```
public class Book {

    private String name;

    @Override
    public String toString() {
        if (name == null){
            name = "yes";
        }
        return "Book{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

如上代码，即展示了惰性初始化，在只有在使用toString方法的时候，才会对name字段进行复制，否则它的值就是null；

使用实例初始化

```java
public class Book {
    private String name;

    public static void main(String[] args) {
        Book book = new Book();
        book.name = new String("yes");
    }
}
```

如上代码，就是对引用字段进行实例初始化

其实对于引用类型字段的初始化没有想的那么复杂，无非也就是这四种情况。

继承语法

```java
public class StoryBook extends Book{

}
```

继承是一件很简单的事儿，我们只需要加上extend关键字即可获得基类的所有字段和方法。

在继承了基类之后仍然可以在本来中定义和使用只属于本类的方法和字段。

同时当前类和基本都拥有相同的字段和方法的时候会优先使用当前类的方法和字段，只有在当前类中找不到目标字段或者方法的时候才会去基类（父类）中寻找并调用。

初始化基类

现在存在基类（父类）和导出类（子类）这两个类。当我们创建一个子类对象的时候，这个对象实际上包含了一个父类的子对象，这个子对象和直接使用父类创建的对象是一样的，不过它不是独立的，而实包含在子类对象内部。

对父类子对象的初始化是十分重要的，这种初始化的方式只有一种，就是在子类的构造器中调用父类的构造器来对父类子对象进行初始化，如果我们不手动调用这个构造器，那么Java会自动在构造器中插入对父类构造器的调用。

```java
class Art{
    Art(){
        System.out.println("第一代");
    }
}

class Drawing extends Art{
    Drawing(){
        System.out.println("第二代");
    }

}

public class Cartoon extends Drawing{

     Cartoon(){
        System.out.println("第三代");
    }

    public static void main(String[] args) {
```

```
        Cartoon cartoon = new Cartoon();
    }
}
```

如上代码，其结果为：

第一代
第二代
第三代

从结果中明显可以看出，对于存在继承的类，他们的构建过程是从基类向外扩散的。所以在子类可以访问父类之前，父类（基类）就已经完成了初始化。

初始化基类——带参数的构造器

上面的情形都是没有带参数的构造器（也就是默认构造器），但是如果父类的构造器是自定义的，有参数的构造器的时候，那么我们继承了父类的子类必须要我们手动的调用父类的构造器。使用super关键字，并且调用语句必须写在子类构造器的第一行。否则编译器就会报错。

```
class Game{
    Game(int i){
        System.out.println("游戏基础");
    }
}

class BoardGame extends Game{

    BoardGame(int i) {
        super(i);
        System.out.println("游戏进阶");
    }
}

public class Chess extends BoardGame{

    Chess(int i) {
        super(i);
        System.out.println("游戏高级");
    }

    public static void main(String[] args) {
        Chess chess = new Chess(12);
    }
}
```

代理

上面知道了Java中类的复用可以使用类的组合和继承两种方式来实现，但是这两种方式也有一些缺点。使用继承的话那子类会拥有父类的所有方法和字段，但是有时为了安全，我们只希望有一部分方法能被子类使用，这个时候就可以使用代理来完成，代理实际上是一个中间人的角色，父类把一些方法交给代理，而在子类中使用代理去调用父类的方法，这样就能达到控制的目的。

```java
/**
 * 太空飞船
 * @author xiaopu
 */
public class SpaceShip {
    String name;
    SpaceShip(String name) {
        this.name = name;
    }
    public static void main(String[] args) {
        SpaceShip spaceShip = new SpaceShip("一号");
    }
}
```

```java
/**
 * 太空飞船控制模块
 * @author xiaopu
 */
public class SpaceShipControls {
    void up(int velocity){}
    void down(int velocity){}
    void left(int velocity){}
    void right(int velocity){}
    void forword(int velocity){}
    void back(int velocity){}
    void turboBoost(int velocity){}
}
```

```java
/**
 * 太空飞船的控制模块代理
 * @author xiaopu
 */
public class SpaceShipDelegation {
    private SpaceShipControls spaceShipControls = new SpaceShipControls();

    public void up(int velocity) {
        spaceShipControls.up(velocity);
    }

    public void down(int velocity) {
        spaceShipControls.down(velocity);
    }

    public void left(int velocity) {
        spaceShipControls.left(velocity);
    }

    public void right(int velocity) {
        spaceShipControls.right(velocity);
    }

    public void forword(int velocity) {
        spaceShipControls.forword(velocity);
    }

    public void back(int velocity) {
```

```
        spaceShipControls.back(velocity);
    }

    public void turboBoost(int velocity) {
        spaceShipControls.turboBoost(velocity);
    }
}
```

上面的例子表明，使用代理的时候，类与类之间没有明确的继承关系，飞船调用控制模块的时候通过控制模块的代理来实现，而不是直接调用控制模块的方法。

继承和组合的结合使用

继承机制和组合机制的混用，是Java开发时候常用的技巧和步骤，虽然编译器会强制提醒我们要初始化父类，但是并不会提醒我们初始化父类的成员对象，因此，这个时候，我们必须时刻注意，避免空指针异常的情况。

正确清理

在Java开发中，我们基本上可以不用管垃圾回收，因为有垃圾回收机制自动帮我们完成这个工作，但是有些特殊的时候，必须要求我们手动的清理垃圾。这个时候我们一定要注意释放顺序，因为在类与类之间可能存在着关系。如果类A正用着类B，如果这个时候我们先释放了B，那么A可能会就报一个空指针异常的错误。

在清理中，最好的办法就是，只把内存管理交给垃圾回收去做。

名称屏蔽：

Java中使用了继承机制的类，子类在重载了父类的方法后，父类的方法仍然是可以使用的。

向上转型：

向上转型是Java继承机制的极为重要的一个特性，通常来讲，Java中有着极其严格的类型检查，但是在继承机制中，子类对象却可以交给父类类型的引用。

```
public class Person {

}

class Student extends Person{
    public static void main(String[] args) {
        Person student = new Student();
    }
}
```

如上代码，student类继承自person类，在创建student类的对象时，可以复制给person类型的变量。

为什么要向上转型：

继承机制是对父类的扩展，所以子类总是比父类更加的具体和详细，而向上转型就是一个较为专用类型向较为通用类型的转换。所以总是很**安全**的，其次，我们在开发过程中，通用的类型更加易用，而专用的类型的使用空间往往比较狭窄。

方法丢失

注意：在向上转型的过程中容易出现的一个问题就是丢失字段和方法。如下代码所示

```java
public class Person {
    String name;

    public void work() {
        System.out.println("工作");
    }
}

class Student extends Person {

    String studentId;

    public void study() {
        System.out.println("学习");
    }

    public static void main(String[] args) {
        Student student = new Student();
        Person person = new Student();
        student.study();
        student.work();
        person.work();
    }
}
```

student类继承自person类，我们在创建对象的时候，将student类型的对象赋值给了person类型的应用，这个时候就实现了向上转型。但是此时实际上是student类型的person变量就不能调用study方法了，因为在person类型中是没有这个方法的。这就是向上转型过程中方法的丢失。

什么时候适合使用继承

当你确定无疑要使用向上转型的时候，这时你应该考虑使用继承机制，否则其他时候通常不要使用继承。

final关键字

final类

被final关键字修饰的类就是fianl类，final类无法被继承。final类的所有方法在默认情况下都会是final级别，因为类都无法被继承，方法自然也无法被继承。但是fianl类的域可以根据自己意愿选择是否为final。

```java
public final class FinalTest {
}
```

final方法

被final修饰的方法就是final方法，用final修饰方法的原因无非两个，第一就是将方法锁定，让其无法被继承以防止任何继承类修改它的含义。第二是出于效率的考虑（解释暂无）

final和private关键字

类中所有的private方法默认情况下都是final级别的，也就是无法被继承，因为private关键字将方法的权限限定在了当前类中，其他类是无法直接访问这个方法的。

这里有一点容易造成疑惑，那就是即使private方法是fianl级别的，表示它无法别继承，但是如果我们继承这个类，我们发现，在子类中**似乎**仍然可以覆盖这个方法，注意这里其实并不是覆盖，而只是在子类中创建了一个和父类有着相同名字和参数列表的方法。它并不是从父类中继承而来的。

覆盖只有某一个方法是父类接口的一部分时才能称之为覆盖，也就是说，如果子类继承了父类，那么在创建对象的时候能够向上转型并且调用到父类的原始方法。但是很明显，在子类中是绝对无法调用到父类中的private方法的

final数据

final数据有两种形式

- 以基本数据类型修饰的数据
- 以引用类型修饰的数据

以基本数据类型修饰的数据用来表示常量，而且和static关键字联合使用，定义时必须对其进行初始化。

```
public static final double PAI =3.14;
```

带有static和final的编译时常量的命名应该全部使用大写，字与字之间使用下划线分开。

对于基本数据类型的final，它的值在程序中是不能改变的。除非去定义它的地方进行修改。

引用数据类型的final则表示的是这个引用的地址不能改变，也就是说一旦引用类型的final指向了一个对象，就不能改变而指向另外一个对象了。但同时final指向的这个引用虽然不能改变，但是引用指向的这个对象本身里面的内容是可以改变的。

注意：我们不能因为某个数据是final的，就认为在编译期间就能知道它的值。实际上他的值是可以在运行时给出的。

```
static Random random = new Random(20);
public static final double PAI = random.nextDouble();
```

这种情况下，PAI的值只有在运行时才被确定。

但是普遍情况下我们使用定义时就赋值的常量的情况较多。

一个即使static又是final的域占用一段不能改变的存储空间。

Java:I/O系统

5个类，3个接口

类	说明
File	文件类
InputStream	字节输入流
OutpuStream	字节输出流
Reader	字符输入流
Writer	字符输出流
Closeable	关闭流接口
Flushable	刷新流接口
Serializable	序列化接口

建议使用的路径方式：I/IDEA_project2/demo/test.java　使用正斜杠的方式

File的三种状态：1.文件 2.文件夹　3.不存在

相对路径和绝对路径：

1. 有盘符的就是绝对路径
2. 没有盘符的就是相对路径

File类：

创建目录：

1. mkdir() : 确保上级目录存在，不存在创建失败
2. mkdirs(): 上级目录可以不存在，不存在一同来创建

列出下一级：

- 1、list() :列出下级名称
- 2、listFiles():列出下级File对象
- 3、列出所有的盘符: listRoots()

获取名称、路径

1. getName() ：获取名称
2. getPath() ：获取路径，创建的时候是相对路径，则返回的就是相对路径，创建的时候是通过绝对路径创建的，那返回的就是据对路径
3. getAbsolutePath() :一定返回绝对
4. getParent(): 返回上一层的路径，如果没有就返回null

文件状态：

1. exists：判断是否存在
2. 若存在的通过isFile判断是否是文件，通过isDirectory判断是否是文件夹（目录）

文件大小：

length()：判断文件的大小，不能判断目录的大小

文件的创建和删除：

1. createNewFile() ：不存在的时候才创建，如果已经存在对应的文档，那么创建失败
2. delete()：删除已经存在的文件

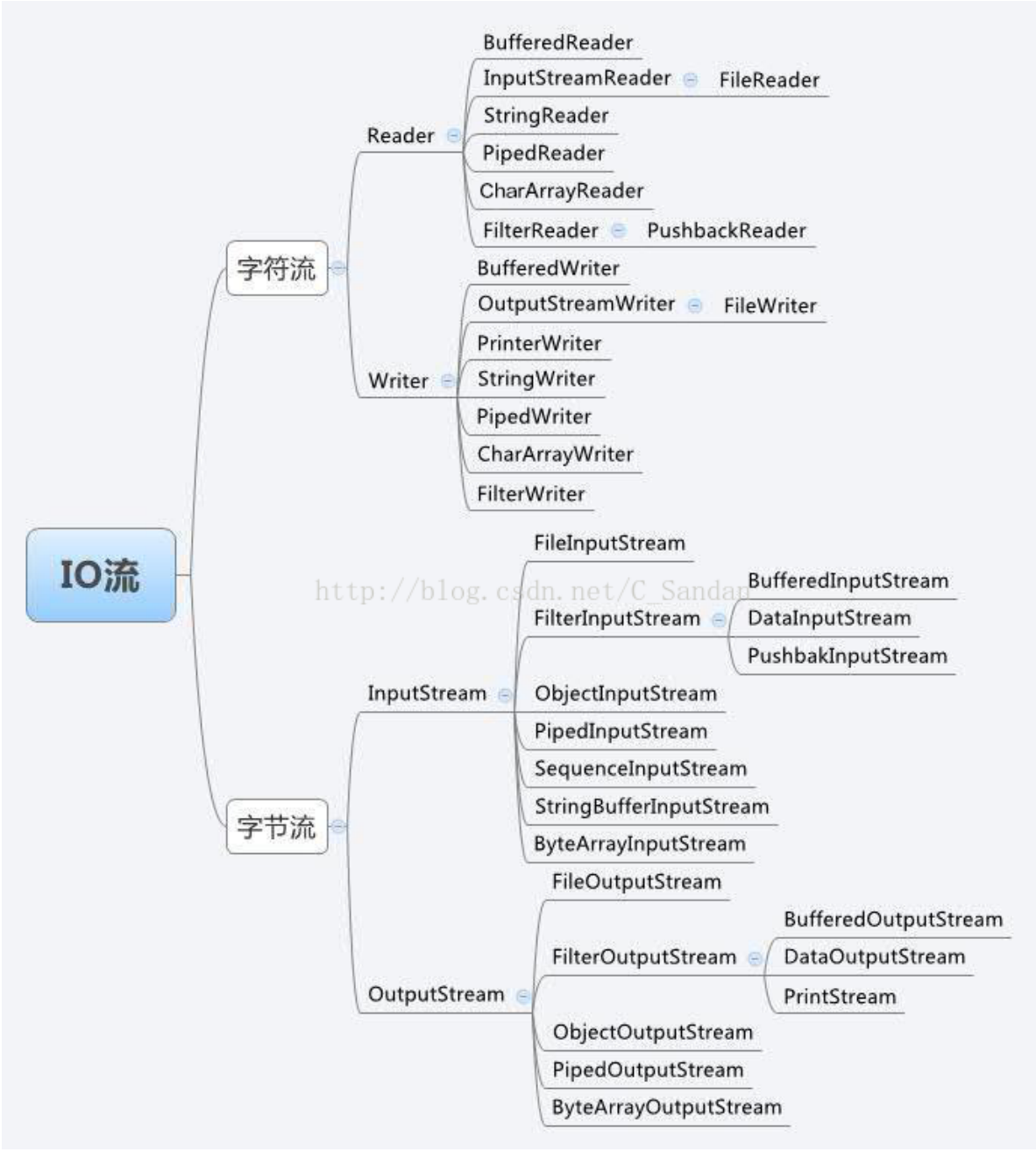
I/O操作基本步骤：

1. 创建数据源
2. 选择流（输入流还是输出流，这两种中的具体又是哪一种）
3. 具体的I/O操作
4. 释放资源

```
public class demo02 {
    public static void main(String[] args) {
        String projectPath = System.getProperty("user.dir");
        File file = new File(projectPath + "/src/main/resources/test.txt");
        InputStream inputStream = null;
        try {
            int temp;
            inputStream = new FileInputStream(file);
            while ((temp=inputStream.read()) != -1){
                System.out.println((char)temp);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            if (inputStream != null){
                try {
                    inputStream.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

以上代码为一个具体的I/O操作实例

I/O流



以程序为中心，向程序中输入数据的称为输入流，从程序中向外输出的称为输出流。

四大抽象类.

抽象类	说明	常用方法
InputStream	字节输入流的父类，数据单位为字节	int read() void close()
OutputStream	字节输出流的父类，数据单位为字节	void write(int) void flush() void close()
Reader	字符输入流的父类，数据单位为字符	int read() void close()
Writer	字符输出流的父类，数据单位为字符	int write(String) void flush() void close()

对于视频、音频等的数据只能使用字节流，而对于文本信息这些则可以直接使用字符流。凡是使用字符流的数据都可以使用字节流，但是反之则不一定可以。

节点流

文件中

FileInputStream（字节流）

文件输入字节流，从文件中读取字节数据。

常用方法

方法	说明
read()	读取一个字节的数据
read(byte[] b)	从文件中读取指定长度的字节到字节数组中

FileOutputStream（字节流）

文件输出字节流，输出字节到文件中。

对于OutputStream中文件路径，如果不存在指定文件，当他写出数据的时候会自动创建这个文件。

常用方法：

方法	说明
write(byte[] b)	将字节数组b大小的字节全部写入文件输出流
write(byte[] b, int off, int len)	将 `len` 字节从指定的字节数组开始，从偏移量 `off` 开始写入此文件输出流。
write(int b)	将指定的字节写入此文件输出流。

FileReader（字符流）

文件字符流，只能处理文本类的字符文件，视频音频等文件无法处理

常用方法

方法	说明
read()	读取一个字符
read(char[] cbuf)	将字符读如数组
read(char[] cbuf, int off, int len)	从指定位置读取指定长度的字符

FileWriter（字符流）

文件字符输出流

常用方法

方法	说明
write(String str)	写一个字符串
write(String str, int off, int len)	写一个字符串的一部分
write(int c)	写一个字符
write(char[] cbuf)	写一个字符数组
write(char[] cbuf, int off, int len)	写一个字符数组的一部分
append(CharSequence csq)	将字符序列追加到指定目标后

内存中

因为字节流是从内存中存取数据，所以尽量不要使用过大的缓冲数组，同时由于字节数组在内存中，所以不用关闭流，垃圾回收器会自动回收。

ByteArrayInputStream（字节流）

字节数组输入流，从内存中输入

常用方法

方法	说明
read()	读取一个字节数据
read(byte[] b, int off, int len)	读取指定长度的数据

ByteArrayOutputStream（字节流）

字节数组输出流，不需要传入数据源，由内存自动分配。

常用方法

方法	说明
write(byte[] b, int off, int len)	写指定长度的数据
write()	写指定的字节的数据
toByteArray()	获取写到内存的数据

处理流

处理流是对前面节点流的封装，提供了更好的性能，使用方法和前面的基本一致；

BufferedInputStream（字节流）

字节输入缓冲流

```
BufferedInputStream(InputStream in)
```

创建时传入输入流对象，用法和前面基本一致

BufferedOutputStream（字节流）

字节输出缓冲流

```
BufferedOutputStream(OutputStream out)
```

创建时传入输出流对象，用法和前面基本一致

BufferedReader（字符流）

字符输入缓冲流

常用方法

方法	说明
readLine()	读取一行，会自动识别换行符

BufferedWriter（字符流）

字符输出缓冲流

常用方法

方法	说明
write(String s)	写一个字符串，此方法继承自父类
newLine()	写一个换行符

转换流

使用方法参照jdk文档，基本和上面的使用方法一致。

InputStreamReader

字节流到字符流的转换桥梁，将指定的文件中的字节还原为字符，相当于解码操作，前提是文件中的数据能够转为字节，如音频视频等则无法完成此操作。

outputStreamWriter

字符流到字节流转换的桥梁，将指定的字符数据写为字节数据，相当于编码操作，有些疑问在与，为什么写为字节了还能看到中文，原因在与看的时候用了对应格式的字符集右对齐进行了解码操作。实际上本质目的是达到了的。

数据流

主要操作基本数据类型和字符串，先写出后读取，读取顺序和写出顺序一致。

DataOutputStream

数据输出流

返回值	方法	描述
`void`	`flush()`	刷新此数据输出流。
`int`	`size()`	返回计数器的当前值 `written`，到目前为止写入此数据输出流的字节数。
`void`	`write(byte[] b, int off, int len)`	将从偏移量 `off` 开始的指定字节数组写入 `len` 字节到底层输出流。
`void`	`write(int b)`	将指定的字节（参数 `b` 的低8位）写入底层输出流。
`void`	`writeBoolean(boolean v)`	将底层输出流写入 `boolean` 作为1字节值。
`void`	`writeByte(int v)`	将底层输出流作为1字节值写入 `byte` 。
`void`	`writeBytes(String s)`	将字符串作为字节序列写入基础输出流。
`void`	`writeChar(int v)`	将底层输出流写入 `char` 作为2字节值，高位字节。
`void`	`writeChars(String s)`	将字符串写入底层输出流作为一系列字符。
`void`	`writeDouble(double v)`	双参数传递给转换 `long` 使用 `doubleToLongBits` 方法在类 `Double` ，然后写入该 `long` 值基础输出流作为8字节的数量，高字节。
`void`	`writeFloat(float v)`	使用 `int` 中的 `floatToIntBits` 方法将float参数转换为 `Float` ，然后 `int` 值作为4字节数量，高字节优先写入底层输出流。
`void`	`writeInt(int v)`	将底层输出流写入 `int` 作为四字节，高位字节。
`void`	`writeLong(long v)`	将底层输出流写入一个 `long` 作为八字节，高字节优先。
`void`	`writeShort(int v)`	将底层输出流写入 `short` 作为两个字节，高字节优先。
`void`	`writeUTF(String str)`	使用机器无关的方式使用 modified UTF-8编码将字符串写入底层输出流。

DataInputStream

数据输入流

返回值	方法	描述
`int`	`read(byte[] b)`	从包含的输入流中读取一些字节数，并将它们存储到缓冲器阵列 `b` 。
`int`	`read(byte[] b, int off, int len)`	从包含的输入流读取最多 `len` 个字节的数据到字节数组。
`boolean`	`readBoolean()`	见 `readBoolean` 方法 `DataInput` 的一般合同。
`byte`	`readByte()`	见 `readByte` 方法 `DataInput` 的一般合同。
`char`	`readChar()`	见 `readChar` 法 `DataInput` 的一般合同。
`double`	`readDouble()`	见 `readDouble` 方法的一般合同 `DataInput` 。
`float`	`readFloat()`	见 `readFloat` 法 `DataInput` 的一般合同。
`void`	`readFully(byte[] b)`	见 `readFully` 法 `DataInput` 的一般合同。
`void`	`readFully(byte[] b, int off, int len)`	见 `readFully` 法 `DataInput` 的一般合同。
`int`	`readInt()`	见 `readInt` 法 `DataInput` 的一般合同。
`long`	`readLong()`	见 `readLong` 方法 `DataInput` 的一般合同。
`short`	`readShort()`	见 `readShort` 法 `DataInput` 的一般合同。
`int`	`readUnsignedByte()`	见 `readUnsignedByte` 法 `DataInput` 的一般合同。
`int`	`readUnsignedShort()`	见 `readUnsignedShort` 法 `DataInput` 的一般合同。
`String`	`readUTF()`	见 `readUTF` 法 `DataInput` 的一般合同。
`static String`	`readUTF(DataInput in)`	从流 `in` 读取以modified UTF-8格式编码的Unicode字符串的表示; 这个字符串然后作为 `String` 返回。
`int`	`skipBytes(int n)`	见 `skipBytes` 法 `DataInput` 的一般合同。

对象流

在数据流的基础上添加了读取和写入一般对象的方法。同样遵循先写后读。读取顺序要和写入顺序一样，同时只有在对象实现了Serializable接口的情况下这个类才可以被写和读，否则无法读写。

如果对于某个敏感字段，不希望被写或者读。则在声明字段的时候加上 transient 关键字

将对象写入也叫序列化或持久化，将对象读取，也叫反序列化。

ObjectOutputStream

对象写出流，用法同上面数据写出流，新增方法，对象写出

ObjectInputStream

对象读取流，用法同上面数据读取流，新增方法对象读取

```
public class Test15 {
    public static void main(String[] args) throws IOException {
        ByteArrayOutputStream byteArrayOutputStream = new
ByteArrayOutputStream();
        BufferedOutputStream bufferedOutputStream = new
BufferedOutputStream(byteArrayOutputStream);
        ObjectOutputStream out = new
ObjectOutputStream(bufferedOutputStream);
        out.writeUTF("你好世界");
        out.writeBoolean(true);
        out.writeChar('c');
        out.writeFloat(1.23f);
        out.flush();
        byte[] bytes = byteArrayOutputStream.toByteArray();

        BufferedInputStream bufferedInputStream = new
BufferedInputStream(new ByteArrayInputStream(bytes));
        ObjectInputStream in = new ObjectInputStream(bufferedInputStream);
        String s = in.readUTF();
        boolean b = in.readBoolean();
        char c = in.readChar();
        float v = in.readFloat();
        System.out.println(b);
    }
}
```

打印流

- PrintStream
- PrintWriter

打印流的特点：

- 不负责数据源，只负责数据目的
- 为其他输出流，添加功能
- 永远不会抛出IO异常，但是可能抛出别的异常

PrintStream

平时用的System.out.println();就是属于这个流；

```
PrintStream out = System.out;
out.println("hello");
```

根据它的构造函数，我们可以很简单的就实现对信息的打印，同时也可以将内容输出到文件中。

可以修改System.out的输出目的地

```
PrintStream ps = new PrintStream(new BufferedOutputStream(new
FileOutputStream("a.txt", true)));
System.setOut(ps);
System.out.println("你好实际");
ps.flush();
```

也可以修改回来

```
PrintStream ps = new PrintStream(new BufferedOutputStream(new
FileOutputStream(FileDescriptor.out)));
System.setOut(ps);
System.out.println("你好实际");
ps.flush();
```

PrintWriter

printWriter和printStream相比在于它多了一个可以接收字节输出流的构造方法，因此它可以对满足对字节输出流的操作。其他的使用方法和printStream没有区别

随机流

RandomAccessFile

该类可以同时支持读取和写出操作

构造方法	描述
`RandomAccessFile(File file, String mode)`	创建一个随机访问文件流，从`File`参数指定的文件读取，并可选地写入。
`RandomAccessFile(String name, String mode)`	创建随机访问文件流，以从中指定名称的文件读取，并可选择写入文件。

从构造方法中，可以看出，第二个参数为mode，这里就是指定其为读（r）模式还是读写（rw）模式的地方。

重要方法

返回值	方法	说明
`void`	`seek(long pos)`	设置文件指针偏移，从该文件的开头测量，发生下一次读取或写入。

使用此方法可以明确的指定从那个位置进行读取。默认情况是指定起始位置，然后读取剩下的全部内容。如果要进行分块读取。则需要自己指定一个块儿大小，然后每次读取的时候就要进行比较，如果剩下的大小>块儿大小，则全部读取，反之则只读取剩下部分大小。

static关键字的含义

static关键字表示为静态的意思，在Java中，被static关键字所标识的内容是直接所属于类的，不属于任何一个具体的对象，当然对于类所属的对象可以调用这个值。不过实际开发中通常都建议直接使用类名来调用。

this关键字和static关键字是不能同时存在来修饰同一个内容的，原因在于this指向的是当前对象的引用，而static关键字根本就没有对象这个说法，所以他们两个这辈子都很难在一起。

对于类中的某个字段来讲，如果他被static修饰，那么这整个类中，这个字段的数据只有一份，而不管创建了多少这个类的对象。因为每一个static字段都只有一份存储空间，而对象则是new一个就会重新开辟一份空间。

Java多线程

多线程是Java语言的天生优势，Java在语言上就支持多线程开发。

简单说明一下程序、进程、线程之间的关系

程序、进程与线程

1. 程序：实际上，在我们的电脑上使用的各种软件就是程序，QQ、LOL，开发工具idea，这些统统都叫程序
2. 进程：当上面说的程序真正运行起来的时候，程序就变成了电脑上的进程。我们可以通过任务管理器查看到
3. 线程：一个程序中包含多个线程，比如一个视频播放器在运行的时候同时调用了视频节码和声卡，他们两个都在一起运行，这就是线程的功劳。又比如说，在浏览器中打开了多个网页，这时候其实一个网页就是一个线程。

Java中默认情况下是有线程存在的，当运行Java程序的时候，main函数执行的其实就是一个线程，叫做主线程，其次如果垃圾回收器运行了，那么还会有gcc线程。

线程的创建

创建方式

1. 继承Thread类（重点）
2. 实现Runnable接口（重点）
3. 实现Callable接口

使用继承Thread类的方式创建线程

使用继承Thread类的方式首先应该继承Thread类，然后覆盖父类的run方法。run方法中就是你的逻辑代码，其次，如果需要开启这个线程，那么应该创建当前类的对象，并调用start()方法，记住是start方法而不是run方法。调用run方法并不会开启线程，而是只会想单线程一样执行代码逻辑。

一下代码运行后，会同时启动两个线程，一个主线程，一个就是因为调用了start方法而开启的子线程，在控制台中打印的时候会出现交替执行的现象。但是我们无法人为控制那个先执行，那个后执行。调度任务交给CPU来完成。

```
1  public class MyThread extends Thread {
2
3      @Override
4      public void run() {
5          for (int i = 0; i < 100; i++) {
6              System.out.println("我在看代码");
7          }
8      }
9
10     public static void main(String[] args) {
11         MyThread myThread = new MyThread();
12         myThread.start();
13         for (int i = 0; i < 100; i++) {
14             System.out.println("我在----主----线程");
15         }
16     }
17 }
```

使用实现Runnable接口的方式创建线程

使用实现Runnable接口的方式，也需要实现run方法，并且把逻辑代码写到run方法内部。但是在启动线程的时候会有所不同，这里仍然需要创建Thread类的对象，然后将实现了Runnable接口的类对象作为参数放入创建Thread类的构造器方法中。最后通过Thread类的对象来启动线程。

```
public class MyRannable1 implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("子线程跑==-=-==-=="+i);
        }

    }

    public static void main(String[] args) {
        MyRannable1 myRannable1 = new MyRannable1();
        //仍然使用thread类对象启动线程
        new Thread(myRannable1).start();
        for (int i = 0; i < 1000; i++) {
            System.out.println("主线程"+i);
        }
    }
}
```

这两种方法中推荐使用Runnable接口的方式。原因在于，使用Runnable接口可以很轻松的让多个线程操作同一份资源，但是在Thread类中实现起来会比较复杂。

使用实现Callable接口创建多线程

使用callable有几个好处，1.能获取结果 2.会获取异常

使用callable接口时，主要分为四个步骤

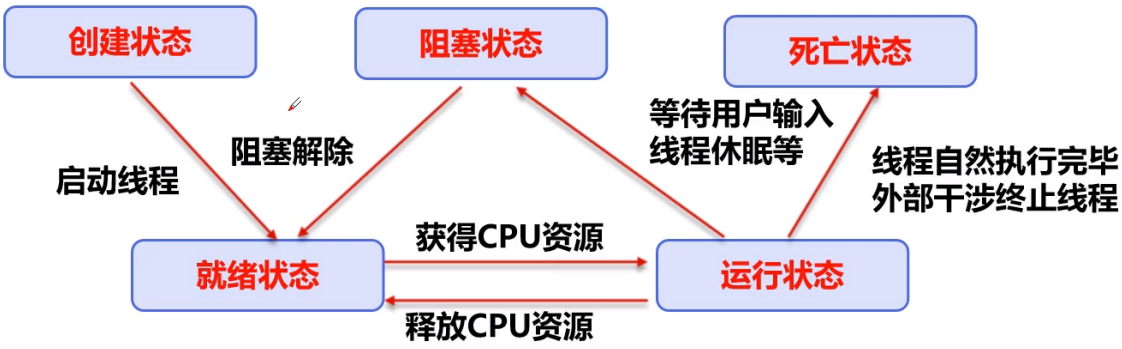
- 创建执行服务
- 提交执行
- 获取结果
- 关闭资源

```
public class MyCallable implements Callable<Boolean> {

    String name;
    MyCallable(String name){
        this.name = name;
    }

    @Override
    public Boolean call() throws Exception {
        for (int i = 0; i < 10; i++) {
            System.out.println(name);
            Thread.sleep(300);
        }
        return true;
    }

```

```
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        MyCallable t1 = new MyCallable("张三");
        MyCallable t2 = new MyCallable("里斯");
        MyCallable t3 = new MyCallable("轨道");

        //创建执行服务
        ExecutorService threadPool = Executors.newFixedThreadPool(3);
        //提交执行
        Future<Boolean> res1 = threadPool.submit(t1);
        Future<Boolean> res2 = threadPool.submit(t2);
        Future<Boolean> res3 = threadPool.submit(t3);
        //获取结果
        Boolean aBoolean1 = res1.get();
        Boolean aBoolean2 = res2.get();
        Boolean aBoolean3 = res3.get();
        //关闭服务
        threadPool.shutdown();

    }

}
```

线程的状态

- 创建状态：当new一个Thread对象的时候，线程就被创建了
- 就绪状态：当调用了start()方法之后，线程就进入了就绪状态，此时需要CPU分配资源才能进入运行状态
- 运行状态：处于就绪状态的线程得到了CPU的资源
- 阻塞状态：等待用户输入等情况，比如让线程睡眠
- 死亡状态：线程执行完毕，或者以外退出了



线程常用方法

方法	描述
setPriority(int)	更改线程优先级
sleep(long)	让当前线程休眠指定秒数
join()	等待该线程停止
yield()	停止当前正在执行的线程对象，并执行其他线程
interrupt()	中断线程
isAlive()	测试线程是否处于活动状态

线程的停止

不推荐使用jdk提供的stop()、destory()等方法【已废弃】

如果能让线程正常的走完生命周期是最理想的停止方式，如果是需要手动停止的话，最好使用标志位的方式来停止，也就是设置一个flag，当为某一种方式的时候就停止线程。

使用标识位停止线程需要注意两个要点

1. 设立一个标识位，一般可以用Boolean类型
2. 提供一个外部方法，用来改变标志位，让线程在指定的位置停止

如下展示使用标志位停止线程的方式

```
public class StopThreadTest implements Runnable {

    /**
     * 设立一个标志位
     */
    private boolean flag = true;

    /**
     * 提供一个外部方法改变标识位
     */
    public void stop(){
        this.flag = false;
    }

    @Override
    public void run() {
        while (flag){
            System.out.println("子线程===");
        }
    }


    public static void main(String[] args) {
        StopThreadTest threadTest = new StopThreadTest();
        new Thread(threadTest).start();

        for (int i = 0; i < 100; i++) {
            System.out.println("===主线程=="+i);
            if (i == 90){
                threadTest.stop();
```

```
                System.out.println("子线程停止");
            }
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

线程的休眠

让线程休眠最常用的方法就是使用Thread.sleep()方法。sleep时间到达之后，线程就又会进入就绪状态，等待CPU分配资源。

每一个对象都有一把锁，sleep不会释放锁

线程礼让(yield()方法)

礼让线程，就是让当前正在执行的线程暂停，但不阻塞，当前线程从运行状态转为就绪状态，所有的线程重回统一起跑线，让CPU重新分配资源。

注意：礼让线程不一定成功，因为都是统一起跑线，所以原来礼让的线程可能又再次获得资源进入运行状态。完全取决于CPU的调度

比如说，A、B两个线程，A处于运行，B处于就绪。现在A礼让线程，两者都回到就绪状态。此时CPU重新分配资源，可能还是A得到资源进入运行，这时就礼让失败了。如果B此时得到资源，那么B就进入了运行。此时礼让成功。

如下代码演示了线程礼让（主要就是礼让方法的演示）

```
public class ThreadYield implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+"开始");
        //线程礼让
        Thread.yield();
        System.out.println(Thread.currentThread().getName()+"结束");
    }

    public static void main(String[] args) {
        ThreadYield threadYield = new ThreadYield();
        new Thread(threadYield,"a").start();
        new Thread(threadYield,"b").start();
    }
}
```

线程强制执行（join()方法）

Join合并线程，只能当插入进来的这个线程执行完毕后其他线程才能执行。和插队一个原理。

```
public class ThreadJoin implements Runnable {
    @Override
    public void run() {
```

```
        for (int i = 0; i < 100; i++) {
            System.out.println("*心*悦*会*员*来*了，通*通*闪*开*"+i);
        }
    }

    public static void main(String[] args) throws Exception {
        ThreadJoin threadJoin = new ThreadJoin();
        Thread thread = new Thread(threadJoin);
        thread.start();
        for (int i = 0; i < 200; i++) {
            if (i==100){
                thread.join();
            }
            System.out.println("主线程默默执行"+i);
        }
    }
}
```

线程的状态判断

Java代码中可以使用以下常量判断线程状态

常量	描述
NEW	线程处于新生状态
RUNNABLE	处于运行状态
BLOCKED	被阻塞等待监视器锁定的线程处于此状态。
WAITING	正在等待另一个线程执行特定动作的线程处于此状态
TIMED_WAITING	正在等待另一个线程执行动作达到指定等待时间的线程处于此状态
TERMINATED	已退出的线程处于此状态

注意：线程一旦执行完了就不能再次启动。

以下代码可判断线程状态

```
public class ThreadState implements Runnable{
    @Override
    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println("嘿嘿");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadState threadState = new ThreadState();
        //线程创建
        Thread thread = new Thread(threadState);
        Thread.State state = thread.getState();
        System.out.println(state);
```

```
18
19          //线程开始执行
20          thread.start();
21          state = thread.getState();
22          System.out.println(state);
23
24          while (state != Thread.State.TERMINATED){
25              Thread.sleep(100);
26              state = thread.getState();
27              System.out.println(state);
28          }
29      }
30  }
```

线程的优先级

线程的优先级用数字表示，范围1~10。优先级高的线程自然容易优先执行。注意：这里说的是容易，所以并不是优先级高的就一定优先执行，有时候并不是这样的。只是优先级高就更加容易先执行。

使用下面两种方法获取或者改变优先级

- getPriority() : 获得当前线程的优先级
- setPriority(int) ： 设置线程的优先级

如下代码演示了线程优先级的启动，记住一点，线程的优先级一定是要先设置，再启动，否则无效。

```
1   public class ThreadPriority implements Runnable{
2       @Override
3       public void run() {
4           System.out.println(Thread.currentThread().getName()+"启动--
    >"+Thread.currentThread().getPriority());
5       }
6
7       public static void main(String[] args) {
8           ThreadPriority threadPriority = new ThreadPriority();
9           Thread t1 = new Thread(threadPriority,"线程1");
10          Thread t2 = new Thread(threadPriority,"线程2");
11          Thread t3 = new Thread(threadPriority,"线程3");
12          Thread t4 = new Thread(threadPriority,"线程4");
13
14          System.out.println(Thread.currentThread().getName()+"启动--
    >"+Thread.currentThread().getPriority());
15
16          t1.setPriority(2);
17          t1.start();
18
19          t2.setPriority(Thread.MAX_PRIORITY);
20          t2.start();
21
22          t3.setPriority(6);
23          t3.start();
24
25          t4.setPriority(8);
26          t4.start();
27      }
28  }
```

守护线程

线程分为用户线程和守护线程，其实在Java开发的时候用户线程很常见，常用的main线程就是用户线程，但是守护线程就不是那么常见了，但是它确实存在。如gc线程（垃圾回收线程），虽然在开发的时候我们无法明确的感知它的存在，但是它却在暗处守护着程序的执行。这就是守护线程。

当两种线程同时存在的时候，虚拟机必须等到用户线程执行完毕的，但是虚拟机却不用等到守护线程执行完毕。

设置守护线程的方法

```
thread.setDaemon(true)
```

线程同步

线程同步发生多个线程操作同一个资源的时候，也就是并发。

要解决并发过程中的同步问题就需要让线程来排队，一个一个来访问，就不会导致资源的泄露和错误。

当然，当队列排好了之后，为了防止某些祸乱分子的捣乱，需要在一个线程在执行的时候，给他提供安全的环境，于是，锁就出现了。当线程获得锁的时候，只有在这个线程完成它的任务后，其他线程才能继续完成他们的任务。

所以要实现线程同步就需要：队列+锁

让线程变得安全

同步方法

同步方法在方法定义上加上synchronized关键字即可让方法变为同步方法。此时的方法只有线程在获得锁之后才能使用。

同步方法默认锁定this，就是调用这个方法的对象本身。

```
public class TicketTest implements Runnable {

    private int ticket = 100;
    private boolean flag = true;

    public synchronized void  buy() throws InterruptedException {
        if(ticket < 1){
            flag = false;
            return;
        }
        Thread.sleep(500);
        System.out.println(Thread.currentThread().getName()+"拿到了第："+
(ticket--)+"张票");
    }

    @Override
    public void run() {
        while (flag){
            try {
                this.buy();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
    }

    public static void main(String[] args) {
        TicketTest ticket = new TicketTest();
        Thread liming = new Thread(ticket, "李明");
        Thread dakang = new Thread(ticket, "达康书记");
        Thread huangniu = new Thread(ticket, "黄牛");

        dakang.start();
        liming.start();
        huangniu.start();
    }
}
```

同步块儿

同步代码块儿就是拥有synchronized关键字的代码块儿，同步代码块儿的obj可以是任意对象，但是推荐使用共享资源作为同步监视器，也就是需要增删改的对象，obj就是同步监视器。

```
synchronized (this){//obj
    if(ticket < 1){
        flag = false;
        return;
    }
    Thread.sleep(500);
    System.out.println(Thread.currentThread().getName()+"拿到了第："+(ticket--)+"张票");
}
```

死锁

死锁发生在多个线程各自占有一些共享资源，并且相互等待其他线程释放占有的资源才能运行，而导致两个或者多个线程都在等待对方释放资源，都停止执行的情形。

某一个同步块同时拥有两个以上对象锁的时候，就可能发生死锁。

死锁产生的四个必要条件

- 一个资源一次被多个线程使用
- 一个线程因为请求资源而被阻塞的时候，其他线程抱着这个资源不放
- 线程已经获得的资源在未使用完之前，无法被强行剥夺
- 若干线程之间形成一种头尾相接，循环等待资源的关系

只要破坏四个必要条件其中之一，就能解除死锁

以下代码演示了死锁的情况，当两个人都在拥有自己的东西的时候有希望获得别人的东西，则就会产生死锁，只有把自己的东西用完了就放开，才能解除死锁。

```
public class Deadlock {
    public static void main(String[] args) {
        Makeup libai = new Makeup(0, "李白");
        Makeup wangzhaojun = new Makeup(1, "王昭君");
        libai.start();
        wangzhaojun.start();
    }
```

```
}


class Mirror {
    String name = "梳妆镜";
}

class Lipstick {
    String name = "阿玛尼口红";
}

class Makeup extends Thread {

    private static Mirror mirror = new Mirror();
    private static Lipstick lipstick = new Lipstick();

    private int flag;
    private String name;

    Makeup(int flag, String name) {
        this.flag = flag;
        this.name = name;
    }

    public void makeUp() throws InterruptedException {
        if (flag == 0){
            synchronized (mirror){
                System.out.println(name+"获得了"+mirror.name);
                Thread.sleep(1000);
                synchronized (lipstick){
                    System.out.println(name+"获得了"+lipstick.name);
                }
            }
        }else {
            synchronized (lipstick){
                System.out.println(name+"获得了"+lipstick.name);
                Thread.sleep(1000);
                synchronized (mirror){
                    System.out.println(name+"获得了"+mirror.name);
                }
            }
        }
    }

    @Override
    public void run() {
        try {
            makeUp();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

解决办法，将上述的同步代码块儿分开锁

```
public void makeUp() throws InterruptedException {
```

```
    if (flag == 0){
        synchronized (mirror){
            System.out.println(name+"获得了"+mirror.name);
            Thread.sleep(1000);
        }
        synchronized (lipstick){
            System.out.println(name+"获得了"+lipstick.name);
        }
    }else {
        synchronized (lipstick){
            System.out.println(name+"获得了"+lipstick.name);
            Thread.sleep(1000);
        }
        synchronized (mirror){
            System.out.println(name+"获得了"+mirror.name);
        }
    }
}
```

Lock锁

lock锁和synchronized同步代码块儿的作用是一样的，不过lock锁是显示的声明同步锁。使用lock的时候使用Lock接口的实现ReentrantLock。

lock锁必须放在try catch语句块儿中，解锁的语句必须放到finally语句块儿中。

```
public class TicketTest implements Runnable {

    private Integer ticket = 10;
    private boolean flag = true;
    Lock lock = new ReentrantLock();

    public void buy() throws InterruptedException {
        if (ticket < 1) {
            flag = false;
            return;
        }
        Thread.sleep(500);
        System.out.println(Thread.currentThread().getName() + "拿到了第：" +
(ticket--) + "张票");
    }

    @Override
    public void run() {
        while (flag) {
            try {
                lock.lock();
                buy();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                lock.unlock();
            }
        }
    }

```

```
    public static void main(String[] args) {
        TicketTest ticket1 = new TicketTest();
        Thread liming = new Thread(ticket1, "李明");
        Thread dakang = new Thread(ticket1, "达康书记");
        Thread huangniu = new Thread(ticket1, "黄牛");

        dakang.start();
        liming.start();
        huangniu.start();

    }
}
```

lock锁和synchronized的对比

synchronized	lock锁
隐式锁，出了作用域自动释放锁	显示锁，必须手动上锁和解锁
有代码块锁和方法锁	只有代码块锁
性能更好	性能相对较差

优先使用顺序：Lock>同步代码块>同步方法

线程通信

通信方法

方法	描述
wite()	线程释放锁进入等待状态
notify()	唤醒正在等待的线程
notifyAll()	唤醒所有

解决生产者消费者问题

管程法

```
public class ProductionAndConsumptionIssues {
    public static void main(String[] args) {
        Buffer buffer = new Buffer();
        new Producer(buffer).start();
        new Consumer(buffer).start();
    }
}


/**
 * 生产者
 */
class Producer extends Thread {

```

```
    Buffer buffer;

    Producer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 20; i++) {
            buffer.push(new Chicken(i));
            System.out.println("生产了第" + i + "只鸡");
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}


/**
 * 消费者
 */
class Consumer extends Thread {

    Buffer buffer;

    Consumer(Buffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 20; i++) {
            try {
                buffer.pop();
                System.out.println("消费了第"+i+"只鸡");
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
}

/**
 * 缓冲区
 */
class Buffer {
    //产品容器
    Chicken[] chickens = new Chicken[10];
    //计数器
    int count = 0;

    /**
     * 向容器中放入产品
```

```
     *
     * @param chicken 鸡肉对象
     */
    public synchronized void push(Chicken chicken) {
        if (count == chickens.length) {
            this.notifyAll();
        } else {
            chickens[count] = chicken;
            count++;
            this.notifyAll();
        }
    }

    /**
     * 取出容器中的对象
     */
    public synchronized Chicken pop() throws InterruptedException {
        Chicken chicken = null;
        if (count == 0) {
            this.wait();
        } else {
            count--;
            chicken = chickens[count];
            this.notifyAll();
        }
        return chicken;
    }

}

/**
 * 产品：鸡肉
 */
class Chicken {
    int id;

    Chicken(int id) {
        this.id = id;
    }
}
```

信号灯法

使用标志位的方式判断哪一个线程等待，那一个线程被唤醒

Executor执行器(线程池)

使用Executor可以很好的管理Thread对象，不用自己来决定线程的生命周期。

Executors可以创建不同的线程池

```
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();
```

```
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);
```

```
ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();
```

newFixedThreadPool可以预先执行线程分配，并且可以限制线程数量

在任何线程池中，现有线程在可能的情况下，都会被自动复用。

synchronized详解

synchronized锁定的是一个对象，每一个对象都有一把锁。

```
String string = new String();
public void paly(){
    synchronized (string){
        for (int i = 0; i < 10; i++) {
            System.out.println("hello");
        }
    }
}
```

如上代码，当执行这个方法中的代码的时候，synchronized就会锁定string对象，在使用这把锁的时候任何其他线程都无法执行这段代码。当synchronized代码块儿执行完毕后，会自动释放这把锁，此时其他线程获得这把锁，未获得锁的线程继续排队等待。这种同时只能有一个线程执行的代码的锁叫互斥锁。

锁定的对象可以自己指定，就像上面这种情况，也可以使用下面的写法。

```
public void paly(){
    synchronized (this){
        for (int i = 0; i < 10; i++) {
            System.out.println("hello");
        }
    }
}
```

使用this关键字表示锁定当前对象。同时这种还有一个等价写法，也叫同步方法。

```
public synchronized void paly() {
    for (int i = 0; i < 10; i++) {
        System.out.println("hello");
    }
}
```

同步方法同样锁定的是this对象，锁同样是在代码块执行完成之后会被释放。

如果synchronized是在静态方法上，那么就相当于是锁定了当前类.class这个对象

```
class Ticket{
    public synchronized static void paly() {
        for (int i = 0; i < 10; i++) {
            System.out.println("hello");
        }
    }
}
```

如上这种情况，相当于是锁定了Ticket.class这个对象。等价于如下情况

```
class Ticket {
    public static void paly() {
        synchronized (Ticket.class) {
            for (int i = 0; i < 10; i++) {
                System.out.println("hello");
            }
        }
    }
}
```

一个synchronized代码块儿必定是一个原子操作，也就是说，当一个线程在执行synchronized代码块儿的时候是无法被打断的，只有当他完整执行完成，别的线程才能执行这个代码块儿。

当一个synchronized方法在执行的过程中，非synchronized方法是可以执行的。

同步方法中是否能够调用另一个同步方法呢？如下情况

```
public class Demo2 {

    public synchronized void m1() {
        System.out.println("hello");
        this.m2();
    }


    public synchronized void m2() {
        System.out.println("thank");
    }
}
```

当m1在执行的时候调用了m2方法，从执行流程上说，执行m1的时候，获取了当前对象的锁，再调用m2的时候也需要去申请这把锁，但是发现这把锁就是自己已经拥有的这把锁。所以此种情况下是允许调用的。此种情况下的锁叫做重入锁。

重入锁还有一种情况，就是在子类的同步方法中调用父类的同步方法，此种情形也是允许的。

需要注意的一点是，当synchronized方法中出现异常的时候，锁会被释放。

volatile关键字

```
volatile boolean flag = true;
```

如上有一个字段是这么定义的，在多线程中，如果需要让多个线程同时知道一个字段发生了改变，那么久需要使用volatile关键字。

通常来讲，在线程执行的时候，cpu将内存中的字段信息复制一份加入cpu的缓冲中，然后判断缓冲中字段的值，如果没有添加volatile关键字，那么在内存中的字段值发生改变的时候，缓冲中的值是无法及时更新的，但是加上了volatile关键字，那么在内存值发生改变的时候，就会通知cpu重新刷新值。

volatile关键字可以保证可见性，但是无法保证原子性，所以volatile无法代替synchronized锁，但是volatile的性能却比synchronized高出很多。

字符串

String对象时不可变的，当创建了一个字符串之后如果对字符串进行改变则会重新创建一个字符串对象，将原来的引用指向它。最初的那个字符串对象则丝毫未动，只是我们无法访问到它了。

```
public class Demo01 {

    static public String upCase(String s){
        return s.toUpperCase();
    }

    public static void main(String[] args) {
        String s1 = "abcdefg";
        String s2 = upCase(s1);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

如上代码，s1是原始字符串的引用，指向了"abcdefg"对象所在的地址，在调用upCase方法的时候实际向方法传递了一个s1引用的拷贝版本。当方法执行完了之后打印，发现原始的对象已经变成了大写的。但是打印s1还是小写，这说明s2实际上指向了一个新的字符串对象。而s1指向的则始终未动。

实际上，每次把String对象作为方法的参数时，都会复制一份引用，而该引用所指的对象其实一致待在原来的物理位置上，从未动过。

String对象拥有只读属性，所以任何指向String对象的引用，原则上都无法修改它的值。

String对象使用“+”

原则上讲，基本上每使用一次“+”，所链接起来的String字符串其实都是一个新的对象。

```
@Aspect
@Component
public class PerformanceAspect {

    @Pointcut("execution(*
com.ctbu.spring.aop.test.service.impl.DancePerform.dance(int)) &&
args(num)")
    public void performance(int num) {
    }

    @Before("performance(num)")
    public int sit(int num) {
        System.out.println("就坐" + num);
        num++;
        return num;
    }
}
```

但实际上，当我们使用“+”进行多个字符串的拼接的时候，编译器会自动帮我们引入StringBuffer类，并调用append()方法。因为这样拼接字符串更加的高效。

String类的操作（常用方法）

方法	说明
charAt(int)	返回参数所指处的char字符，参数范围0 ~（length-1）
getChars()	将本字符串复制到一个字符数组中
getBytes	将字符串转换为编码的字节
toCharArray()	生成一个字符数组，包含字符串中所有的字符
equals	将字符串与另一个字符串进行比较，看是否相等。
equalsIgnoreCase	字符串比较，忽略大小写
compateTo	字符串比较
contains	检查字符串是否包含参数中的字符串，是则为true
contentEquals	字符串与参数完全一致则为true
regionMatches	比较两个字符串区域内是否相等
startsWith	是否以某一个字符作为开头
endsWith	是否以某一个字符作为结尾
indexOf	返回匹配字符第一次出现的索引
lastIndexOf	返回指定字符最后一次出现的索引
substring	从指定索引开始，截取到字符串尾
subSequence	从指定的开始索引截取到指定的结束索引
concat	拼接上参数
repalce	将匹配的字符替换为参数中的字符，可以是单个字符的替换，也可以是字符串的替换
toLowerCase	转换为小写
toUpperCase	转换为大写
trim	删除字符串两端的空白，如果没有发生删除操作，则返回原来的对象
valueOf	静态方法，将其他类型转换为string类型
intern	将字符串放入字符串池中

Java中的正则表达式

Java中的反斜杠：在其他语言中，使用 \\ 表示普通意义上的一个反斜杠，也就是字面意思上的反斜杠，而在Java中 \\ 的意思则是表示一个正则表达式的反斜杠，后面紧跟的字符具有特殊的意义。比如表示一位数字，则在Java的正则表达式中应该是 \\d ，如果是插入一个普通的反斜杠，则应该是 \\\\ ，不过仍然有一部分只需要单反斜杠即可完成，比如制表符，换行符之类的。

String类自带的正则表达式方法：

方法	说明
split()	将字符串从正则表达式匹配的地方切开
replace	将指定的字符替换为其他字符，有多个重载版本

正则表达式的语法

非打印字符

字符	描述
\cx	匹配由x指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。 否则，将 c 视为一个原义的 'c' 字符。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。

特殊字符

所谓特殊字符，也就是在正则表达式中起到一定的特殊作用的字符，比如“+”，在正则表达式中它表示前面的表达式重复一次或者多次， 并不是我们平常以为的加号的意思，如果我们匹配的时候正好要匹配的内容中有“+”号，那么就需要对其转义，进而让正则表达式中的特殊字符回归它本来的意思。

特别字符	描述
$	匹配输入字符串的结尾位置。如果设置了 RegExp 对象的 Multiline 属性，则 $ 也匹配 '\n' 或 '\r'。 要匹配 $ 字符本身，请使用 $。
( )	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。要匹配这些字符，请使用 (和)。
*	匹配前面的子表达式零次或多次。要匹配 * 字符，请使用 *。
+	匹配前面的子表达式一次或多次。要匹配 + 字符，请使用 +。
.	匹配除换行符 \n之外的任何单字符。要匹配 .，请使用 \。
[	标记一个中括号表达式的开始。要匹配 [，请使用 [。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。要匹配 ? 字符，请使用 \?。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如，'n' 匹配字符 'n'。'\n' 匹配换行符。序列 '\' 匹配 ""，而 '(' 则匹配 "("。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。 要匹配 ^ 字符本身，请使用 \^。
{	标记限定符表达式的开始。要匹配 {，请使用 {。
|	指明两项之间的一个选择。要匹配 |，请使用 |。

次数限定符

次数限定符也就是匹配元素需要重复多少次才能比配成功，同样用“+”举例，'zo+' 能匹配 "zo" 以及 "zoo"，也就是o出现一次或多次。在正则表达式中，一共有*或+或?或{n}或{n,}或{n,m}共6种。

字符	描述
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于{0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 、 "does" 中的 "does" 、 "doxy" 中的 "do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。

定位符

定位符能确定匹配元素在字符串中的位置

字符	描述
^	匹配输入字符串开始的位置。如果设置了 RegExp 对象的 Multiline 属性，^ 还会与 \n 或 \r 之后的位置匹配。
$	匹配输入字符串结尾的位置。如果设置了 RegExp 对象的 Multiline 属性，$ 还会与 \n 或 \r 之前的位置匹配。
\b	匹配一个字边界，即字与空格间的位置。
\B	非字边界匹配。

元字符

字符	描述
\	将下一个字符标记为一个特殊字符、或一个原义字符、或一个 向后引用、或一个八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\' 匹配 "" 而 "(" 则匹配 "("。
^	匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。
$	匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。
*	匹配前面的子表达式零次或多次。例如，zo* 能匹配 "z" 以及 "zoo"。* 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。
?	匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。
{n,}	n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o*'。
{n,m}	m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。
?	当该字符紧跟在任何一个其他限制符 (*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
(pattern)	匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在VBScript 中使用 SubMatches 集合，在JScript 中则使用 $0...$9 属性。要匹配圆括号字符，请使用 '(' 或 ')'。
(?:pattern)	匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用 "或" 字符 (|) 来组合一个模式的各个部分是很有用。例如，'industr(?:y|ies) 就是一个比 'industry|industries' 更简略的表达式。
(?=pattern)	正向预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，'Windows (?=95|98|NT|2000)' 能匹配 "Windows 2000" 中的 "Windows" ，但不能匹配 "Windows 3.1" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。

字符	描述
(?!pattern)	负向预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如'Windows (?!95|98|NT|2000)' 能匹配 "Windows 3.1" 中的 "Windows"，但不能匹配 "Windows 2000" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。
x|y	匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。
[xyz]	字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。
[^xyz]	负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'。
[a-z]	字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。
[^a-z]	负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\cx	匹配由 x 指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 [^0-9]。
\f	匹配一个换页符。等价于 \x0c 和 \cL。
\n	匹配一个换行符。等价于 \x0a 和 \cJ。
\r	匹配一个回车符。等价于 \x0d 和 \cM。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\t	匹配一个制表符。等价于 \x09 和 \cI。
\v	匹配一个垂直制表符。等价于 \x0b 和 \cK。
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'。
\W	匹配任何非单词字符。等价于 [^A-Za-z0-9_]。
\xn	匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，'\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。
\num	匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。例如，'(.)\1' 匹配两个连续的相同字符。

字符	描述
\n	标识一个八进制转义值或一个向后引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。
\nm	标识一个八进制转义值或一个向后引用。如果 \nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果 \nm 之前至少有 n 个获取，则 n 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字 (0-7)，则 \nm 将匹配八进制转义值 nm。
\nml	如果 n 为八进制数字 (0-3)，且 m 和 l 均为八进制数字 (0-7)，则匹配八进制转义值 nml。
\un	匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如， \u00A9 匹配版权符号 (?)。

匹配模式

- **贪婪型**（最大匹配）：例如你要用 “<.+>” 去匹配 “a<tr>aava </tr>abb”，也许你所期待的结果是想匹配 “<tr>”，但是实际结果却会匹配到 “<tr>aava </tr>，在贪婪模式下，会尽可能的扩大匹配范围，当扩大匹配无法成功时，又会缩小匹配范围
- **勉强型**（最小匹配）：也就是只要匹配成功，就不再扩大匹配范围
- **占有型**（完全匹配）：在贪婪型的基础上不会重新缩小匹配范围

贪婪型的后面加上？就是勉强型，加上+就是占有型

贪婪型	勉强型	占有型
X?	X??	X?+
X*	X*?	X*?
X+	X+?	X++
X{n}	X{n}?	X{n}+
X{n,}	X{n,}?	X{n,}+
X{n,m}	X{n,m}?	X{n,m}+

三个类

- **Pattern**类：pattern对象是一个正则表达式的编译表示。Pattern类没有公共构造方法。要创建一个Pattern对象，你必须首先调用其公共静态编译方法，它返回一个Pattern对象。该方法接受一个正则表达式作为它的第一个参数。
- **Matcher**类：Matcher对象是对输入字符串进行解释和匹配操作的引擎。与Pattern类一样，Matcher也没有公共构造方法。你需要调用Pattern对象的matcher方法来获得一个Matcher对象。
- **PatternSyntaxException**类：PatternSyntaxException是一个非强制异常类，它表示一个正则表达式模式中的语法错误。

Pattern类和Matcher类对于Java中的正则表达式来说是很重要的，我们无法直接new处这两个类的对象，一般操作流程如下。

使用静态方法Pattern.compile()参数为正则表达式，他会返回一个Pattern类型的对象，这是一个正则表达式的编译版本，然后再用其调用matcher方法，此方法会生成一个Matcher类型的对象。Matcher类型的对象有很多功能可以使用。例如下面的例子，使用replaceAll方法可以将匹配的部分都替换成传入的参数。

```
public class Demo01 {
    public static Pattern pattern = Pattern.compile("[12]\\d{4,10}@qq\\.com");
    public static void main(String[] args) {
        Matcher matcher = pattern.matcher("ni hao 1182810784@qq.com");
        String s = matcher.replaceAll("****");
        System.out.println(s);
    }
}
```

matches()

Pattern拥有静态方法，matches()，用于判断一个字符串是否能匹配一个正则表达式

```
public class Demo01 {
    public static void main(String[] args) {
        boolean matches = Pattern.matches(".*[12]\\d{4,10}@qq\\.com", "ni hao 1182810784@qq.com");
        System.out.println(matches);
    }
}
```

split()

对于编译后的Pattern对象，还提供了split()方法，支持将字符串从匹配正则表达式的地方切开。

```
public class Demo01 {
    public static Pattern pattern = Pattern.compile("\\.");
    public static void main(String[] args) {
        String[] split = pattern.split("nihao.alottou.chizi");
        for (String s : split) {
            System.out.println(s);
        }
    }
}
```

matcher

对于编译后的pattern对象，通过调用matcher方法后能获得一个Matcher对象，Matcher对象继续使用matches方法查看字符串是否匹配正则表达式。同时能使用lookingAt方法来判断字符串能否部分匹配

```java
public class Demo01 {
    public static Pattern pattern = Pattern.compile("[123]d");
    public static void main(String[] args) {
        Matcher matcher = pattern.matcher("2da");
        System.out.println(matcher.matches());//false
        System.out.println(matcher.lookingAt());//true
    }
}
```

find和group

matcher对象中还有find方法和group方法，find方法可以判断处在字符串中有几处匹配正则表达式的，group方法能获取匹配到的子字符串

```java
public class Demo01 {
    public static Pattern pattern = Pattern.compile("a.b");
    public static void main(String[] args) {
        Matcher matcher = pattern.matcher("adbsldkfjaibsldf");
        while (matcher.find()){
            System.out.println(matcher.group());
        }
    }
}
```

group有重载版本，可以根据不同的组引用获取对应匹配的字符串字串。

start和end

start方法返回匹配的字符串的起始位置字符的索引，end方法返回字符匹配成功的最后一个字符的索引值+1，这两个方法之后在匹配成功之后才能调用，若失败或者在没有匹配的时候去调用则会抛出异常。

```java
public class Demo01 {
    public static Pattern pattern = Pattern.compile("a.b");
    public static void main(String[] args) {
        Matcher matcher = pattern.matcher("dfdadbsldkfjaibsldf");
        while (matcher.find()){
            System.out.println(matcher.start());
            System.out.println(matcher.end());
        }
    }
}
```

replaceFirst和replaceAll

replaceFirst方法会替换掉匹配成功的第一个字符串，替换内容为参数，replaceAll会替换掉所有匹配成功的字符串，替换内容为参数

```java
public class Demo01 {
    public static Pattern pattern = Pattern.compile("a.b");
    public static void main(String[] args) {
        Matcher matcher = pattern.matcher("dfdadbsldkfjaibsldf");
        String s = matcher.replaceFirst("***");
        String s1 = matcher.replaceAll("00000");
        System.out.println(s);
        System.out.println(s1);
    }
}
```

appendReplacement

```java
matcher.appendReplacement(sb, replaceContent);
```

sb是一个StringBuffer，replaceContext待替换的字符串，这个方法会把匹配到的内容替换为replaceContext，并且把从上次替换的位置到这次替换位置之间的字符串也拿到，然后，加上这次替换后的结果一起追加到StringBuffer里（假如这次替换是第一次替换，那就是只追加替换后的字符串啦）。

appendTail

```java
matcher.appendTail(sb);
```

appendTail方法：sb是一个StringBuffer，这个方法是把最后一次匹配到内容之后的字符串追加到StringBuffer中。

reset

reset方法可以将已有的正则表达式应用到一个新的字符串中。

```java
public class Demo01 {
    public static Pattern p = Pattern.compile("cat");

    public static void main(String[] args) {
        Matcher m = p.matcher("one cat two cats in the yard");
        Matcher matcher = m.reset("cats in the yard");
    }
}
```

组

组是用括号表达的正则表达式，可以根据组的编号来引用某个组，组号为 0 的表示整个表达式，组号为1的表示第一对括号括起来的组，以此类推。

如A（B（C））D。拥有三个组，0组：ABCD 1组：BC 2组：C

Matcher对象可以使用groupCount方法获取本正则表达式的分组数，第0组不包括在内

```java
public class Demo01 {
    public static Pattern pattern = Pattern.compile("([123]d).*?");
    public static void main(String[] args) {
        Matcher matcher = pattern.matcher("2dlsd");
        int count = matcher.groupCount();
        System.out.println(count);
    }
}
```

Pattern标记

pattern的compile方法还有一个版本

```java
compile(String regex, int flags)
```

flags可以调整匹配标准

标记	说明
CANON_EQ	设置规范等价
CASE_INSENSITIVE（?!)	不区分大小写的匹配
COMMENTS(?x)	允许注释和空格格式
DOTALL(?s)	点阵模式，表达式会匹配任何字符
MULTILINE(?m)	启动多行模式，表达式`^`和`$分别`匹配行终止符或输入序列的结尾
UNICODE_CASE(?u)	当指定此标志时，不区分大小写的匹配（由`CASE_INSENSITIVE`标志启用）以与Unicode标准一致的方式完成。 默认情况下，不区分大小写的匹配假定仅匹配US-ASCII字符集中的字符。
UNIX_LINES(?d)	在这种模式下，只有`'\n'`行结束在`.`，`^`和`$`行为的认可。

类型信息

Class对象

Class对象就是用来创建类的所有常规对象的。类是程序的一部分，每一个类都有一个Class对象，换言之，每当编译了一个新类，就会产生一个Class对象。

所有的类都是在对其第一次使用时，动态的加载到JVM中的。当程序创建第一个对类的静态成员的引用时，就会加载这个类。故，构造器也是类的静态方法，即使在构造器之前没有添加static关键字。因此，每次使用new关键字创建对象也会被当作时对类的静态成员的引用。

Java程序在运行时，并不是将所有字节码都加载进去，而是在用到的时候才加载。

在Java中，static成员的初始化是在类加载的时候完成的。

Class.forName()

Class.forName()方法是Class类的一个静态方法，此静态方法会返回Class对象的引用。

不管什么时候，如果你想在运行时获得类型信息，首先必须就是要获得Class对象的引用。使用Class.forName()就可以快速的办到这一点，而且他还有个有点，也就是不需要你创建了对应类的对象之后才能获得这个Class对象的引用，即使你没有这个对象，你也可以直接通过其类名获得此引用。（forName的参数是全限定名称，包含包名）

但是如果你已经拥有了类的对象，那么你也可以通过这种方式来获得Class对象的引用。

getClass()

如果你确实已经创建了某一个对象，那么你可以直接使用这个对象调用getClass方法来获取当前类的Class对象的引用。

Class对象的方法

方法	说明
getNmae	获取类包名+类名
getSimpleName	获取不带包名的类名
getCanonicalName	获取全限定类名
isInterface	判断类是否为接口
getInterfaces	获取当前类所实现的所有接口
getSuperclass	获取当前类所继承的类
newInstance	给当前类创建一个新的实例，需要无参构造器

方法	说明
isInstance	判断一个对象能否转化为某一个类的对象
getFields	获取类的属性（只能是public的）
getField(String)	根据属性名获取属性，仅限public
getDeclaredFields	获取所有的属性，包括public、private、protect和默认权限的
getDeclaredField(String)	根据属性名获取属性，所有权限的都可以
getMethods	获取所有的public方法
getMethod(String,Class<?>...)	根据方法名称获取public方法，后面的参数是可变参数，为形参对应类型的Class对象
getDeclaredMethod(String, Class<?>...)	根据方法名称获取任意权限的方法,后面的参数是可变参数， 为形参对应类型的Class对象
getDeclaredMethods	获取所有权限的方法
getConstructor(Class<?>...)	获取public的构造器
getConstructors(Class<?>...)	获取所有的public构造器
getDeclaredConstructor(Class<?>...)	获取所有权限的构造器
getDeclaredConstructors	获取全部所有权限的构造器
getAnnotations	获取类上标注的所有注解
getAnnotation(Class<A>)	根据注解类型获取注解

需要注意的是，使用newInstance创建的类必须带有默认的构造器，创建出来后可以使用某种转型，比如知道创建出来的对象的确切类型，从而可以使用强制转型。

类字面常量

除了上面说的两种方式来获取一个类的Class对象，Java还提供了另外一种方法来生成对Class对象的引用，即字面量常量。

```
1 FancyToy.class
```

使用此种方法更加高效，安全，而且简单。

此种方法可以应用于普通的类、接口、数组、基本数据类型。另外，对于基本数据类型的包装类，有一个字段TYPE，TYPE字段是一个引用，指向的是对应基本数据类型的Class对象。

基本数据类型引用	TYPE字段
int.class	Integer.TYPE
long.class	Long.TYPE
double.class	Double.TYPE

其他的基本数据类型以此类推。

以上三种方式，建议优先使用.class的方式。

使用一个类的过程

- **加载**：由类加载器执行，这一步查找字节码文件，并从这些字节码中创建一个Class对象
- **链接**：此阶段将验证类中的字节码，为静态成员分配存储空间，如果存在对其他类的引用，那么将会解析这个类创建的对其他类的引用
- **初始化**：如果这个类具有父类，则对其进行初始化，执行静态初始化器和静态初始化块

使用。class的方式获取对类的引用不会引发类的初始化操作，但是使用Class.forName()就会立刻触发初始化操作。

static静态代码块

静态代码块会在加载类的时候就执行，而且全过程只执行一次。

static final字段的初始化

如果static final字段的值是一个编译时常量，也就是编写代码时就确定的值，那么这个值是在其所在类不需要初始化的情况下也是可以读取的。但是如果这个字段是运行时的常量，那么就必须是类的初始化之后才能访问。

如果一个static字段不是final的，那么在对它访问的时候，要先进行连接和初始化（分配空间和初始值）

Class的泛化引用

Class的引用总是指向某一个Class对象，使用这个对象可以创建指定类的实例。并且可以通过这个class对象获取这个类的所有字段，方法以及静态成员。因此可以这样理解，Class引用实际上表示的就是它所指向的对象的确切类型。

```
public class GenericClassReferences {
    public static void main(String[] args) {
        Class intClass = int.class;
        Class<Integer> integerClass = int.class;
        integerClass = Integer.class;
        intClass = double.class;
        //integerClass = double.class;//错误
    }
}
```

如上代码中，很容易发现，对于普通的类引用，可以很容易的指向其他Class对象的引用，因为这是允许的。但是如果使用了泛型，那么就不能随意复制，因为存在类型检查。

为了能够放松这类检查，引入了通配符“?"，他表示任何事物。

```
public class GenericClassReferences {
    public static void main(String[] args) {
        Class<?> integerClass = int.class;
        integerClass = double.class;
    }
}
```

此时这个引用就可重新赋值给其他类型的引用。

尽管普通的Class和Class<?>的作用是等价的，但是后者仍然优于前者。

如果你想限制具体类型或者限制继承关系，那么你尽管可以这样使用。

```
public class GenericClassReferences {
    public static void main(String[] args) {
        Class<? extends Number> integerClass = int.class;
        integerClass = double.class;
    }
}
```

加上extend后可以限定这个Class对象的引用范围。

给Class引用添加泛型检查就额外的提供了类型检查，防止出错。

cast方法

```
class Building{}
class House extends Building{}

/**
 * @author xiaopu
 */
public class ClassCasts {
    public static void main(String[] args) {
        Building b = new House();
        Class<House> houseType = House.class;
        House h = houseType.cast(b);
        h = (House) b;
    }
}
```

如上代码，可以很明确的看到，cast方法接收参数为对象实例，它可以将一个泛型对象转换为一个具体类型的对象。这和使用（）的方式实现强制类型转换的效果一样，但是当你要通过Class对象来实现转型的时候，可能会用到这个方法。

instanceof关键字（类型转换前做好类型检查）

使用instanceof关键字可以检查某一个对象类型的引用指向的是否为指定的类型。也就是能够判断一个类型的具体类型是什么。

```java
class Building{}
class House extends Building{}

public class ClassCasts {
    public static void main(String[] args) {
        Building build = new House();
        House h;
        if (build instanceof House){
            h = (House)build;
        }
    }
}
```

在向下转型之前，如果不知道被转型对象的具体类型，那么这个操作是很危险的，这样做很可能会触发异常，所以在转型之前判断一下类型是很有必要的。

注解

注解可以使用在类、方法、字段上，可以对类进行说明，同时也可以被其他程序读取，进而实现更加强大的功能。

内置注解

注解	说明
@Override（方法级）	表示当前方法正在尝试覆盖父类的方法
@Deprecated（方法、属性、类级）	表示开发者最好不要使用这个元素
@SuppressWarnings	忽略编译时出现的警告（有可选参数）

元注解

注解	说明
@Target	表明定义的注解用在什么地方
@Retention	表示在什么级别保存该注解的信息
@Documented	将此注解包含在Javadoc中
@Inherited	允许子类继承父类中的注解

@Target

参数：需要一个参数，用于指定注解用于何处

参数	说明
PACKAGE	用于包
TYPE	用于类
CONSTRUCTOR	用于构造器
FIELD	用于域
METHOD	用于方法
LOCAL_VARIABLE	用于局部变量
PARAMETER	用于属性字段

@Retention

参数：可以限定在什么级别保存注解信息

参数	说明
SOURCE	在云文件中有效
CLASS	在class文件中有效
RUNTIME	在运行时有效，为此值时注解可以被反射机制读取

自定义注解

自定义注解的时候需要元注解的帮助，以此来完善我们自己的注解。同时我们自己也要给注解定义参数等信息。

```
@Target(value = {ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {

    String name() default "";
    int level() default 0;

}
```

如上，我们便定义了一个属于自己的注解，这个注解只能用在方法上，同时在运行时也是保存的。在给注解定义字段的时候有点像接口中方法的定义，但这完全是两个东西，这里明确的是对字段的定义。

需要注意的一点是，注解中的所有字段必须有值，所以我们常常会设置一个默认值，如上的代码所示。

如果定义的字段只有value，那么在使用这个注解的时候，可以不用写字段名value而直接写上字段的值。

反射

每一个类都有且只有一个对应的Class对象，通过Class对象可以获得这个类的所有数据信息，包括方法，字段等数据。

通过反射调用构造器创建对象

使用调用默认构造器

```
Class<Person> aClass = Person.class;
Person person = aClass.newInstance();
```

使用有参构造器

```
Class<Person> aClass = Person.class;
Constructor<Person> constructor = aClass.getDeclaredConstructor(String.class,
int.class);
Person person = constructor.newInstance("李白",23);
```

通过反射调用普通方法

```
Class<Person> aClass = Person.class;
Person person = aClass.newInstance();
Method eat = aClass.getDeclaredMethod("eat");
eat.invoke(person);//第一个参数指定是那个对象调用了此方法，第二个参数是可变参数，如果没有就不写
```

通过反射操作属性

```
Class<Person> aClass = Person.class;
Person person = aClass.newInstance();
Field name = aClass.getDeclaredField("name");
name.set(person,"李白");
System.out.println(person.getName());
```

操作私有方法和属性

很多时候，有可能方法和属性是私有的，特别是属性，我们无法直接操作，这个时候可以使用暴力反射的方式进行操作，也就是在操作之前先将他的安全检查击溃。比如上面的属性操作中，在操作之前name.setAccessible(true);

```
Class<Person> aClass = Person.class;
Person person = aClass.newInstance();
Field name = aClass.getDeclaredField("name");
name.setAccessible(true);
name.set(person,"李白");
System.out.println(person.getName());
```

反射操作泛型

反射对泛型的操作，主要时对方法的参数的确定和返回值泛型的确定。

```java
public class Demo4 {
    public void test01(Map<String, Person> map, List<Person> list){
        System.out.println("Demo4.test01");
    }

    public Map<Integer,Person> test02(){
        System.out.println("Demo4.test02");
        return null;
    }

    public static void main(String[] args) {
        try {
            Class<?> aClass = Demo4.class;
            Method test01 = aClass.getMethod("test01", Map.class,
List.class);
            Type[] types = test01.getGenericParameterTypes();//获取泛型参数类
型
            for (Type type : types) {
                System.out.println(type);
                if (type instanceof ParameterizedType){//判断是否为参数类型
                    Type[] arguments = ((ParameterizedType)
type).getActualTypeArguments();//强制转型、获取实际参数类型
                    for (Type argument : arguments) {
                        System.out.println("泛型类型"+argument);
                    }
                }
            }

            Method test02 = aClass.getMethod("test02");
            Type returnType = test02.getGenericReturnType();
            if (returnType instanceof ParameterizedType){
                Type[] arguments = ((ParameterizedType)
returnType).getActualTypeArguments();
                for (Type argument : arguments) {
                    System.out.println("泛型返回类型: "+argument);
                }
            }
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        }
    }
}
```

反射操作注解

获取类上的注解

1.根据注解类型明确获取

```java
Class<?> aClass = Student.class;
Table annotation = aClass.getAnnotation(Table.class);
```

2.获取类上的所有注解

```
Class<?> aClass = Student.class;
Annotation[] annotations = aClass.getAnnotations();
```

获取属性上的注解

1.根据注解类型获取

```
Field stuName = aClass.getDeclaredField("stuName");
annotation.Field field = stuName.getAnnotation(annotation.Field.class);
```

2.获取字段上的所有注解

```
Field stuName = aClass.getDeclaredField("stuName");
Annotation[] annotations = stuName.getAnnotations();
```

3.获取非public权限的注解

```
Field stuName = aClass.getDeclaredField("stuName");
Annotation[] annotations = stuName.getDeclaredAnnotations();
```

其他的要举一反三

获取方法的注解

```
Class<?> aClass = Student.class;
Method getId = aClass.getMethod("getId");
annotation.Method annotation = getId.getAnnotation(annotation.Method.class);
```

其他获取方式类似

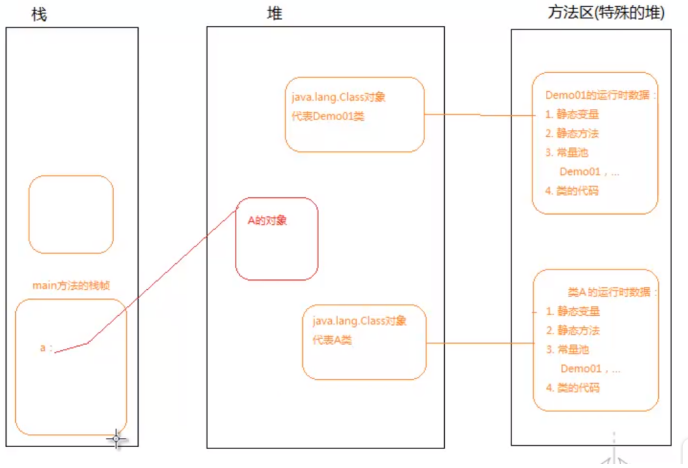
反射中提高运行效率

在反射中提高效率的方法之一就是禁止安全检查

```
setAccessible();
```

类加载过程

```
package com.bjsxt.test;

public class Demo01 {

	public static void main(String[] args) {
		A a = new A();
		System.out.println(A.width);
	}
}


class A {
	public static int width=100;

	static {
		System.out.println("静态初始化类A");
		width=300;
	}
	public A(){
		System.out.println("创建A类的对象");
	}
}
```



当Java程序需要使用某个类时，如果该类还未被加载到内存中，JVM会通过加载、连接(验证、准备和解析)、初始化三个步骤来对该类进行初始化。

类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，读入的数据方法JVM内存的方法区中，然后产生与所加载类对应的Class对象，Class对象放入JVM的堆区中。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备(为静态变量分配内存并设置默认的初始值)和解析(将符号引用替换为直接引用)三个步骤。最后JVM对类进行初始化，包括：

1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类;

2)如果类中存在初始化语句，就依次执行这些初始化语句。

当真正运行代码的时候，会有一个栈区，也叫方法栈，每当调用一个方法的时候就将方法压栈，并在对应方法栈中保存方法中的字段。此时如果在方法中new了一个对象，那么就会调用对应的构造器，创建一个对象，创建的对象放入堆中，并且把应用交给对应方法栈中的字段。

代码的运行在初始化之后，方法的调用在代码运行中，所以，之后先完成了初始化，才能运行代码，才能调用方法。

总结下来就是内存中的三个区域

区域	说明
堆	存放类对应的Class对象、new出来的普通对象
栈	加载方法时将方法压栈并保存方法中对应的字段
方法区（特殊的堆）	存放类的数据结构，包括类的各种静态常量、静态方法、静态变量、类的代码等

初始化的时机

类的主动引用（一定会发生类的初始化）

- new一个类的对象
- 调用类的静态成员（除去final常量）和静态方法
- 使用java.lang.reflect包的方法对类进行反射调用
- 启动了main方法所在的类
- 如果一个类拥有父类且没有被初始化，则会先初始化父类

类的被动引用（不会发生类的初始化）

- 访问一个静态域时，只有真正声明这个域的类才会被初始化（通过子类引用父类的静态变量不会导致子类的初始化）

```
1  public class A {
2      public static void main(String[] args) {
3          System.out.println(C.PAI);
4      }
5  }
6
7  class B {
8      public static double PAI = 3.14;
9
10     static {
```

```
        System.out.println("B初始化了");
    }
}

class C extends B {
    static {
        System.out.println("C初始化了");
    }
}
```

此时C不会初始化，但是B会初始化。

- 通过数组定义类的引用，不会触发类的初始化

```
public class A {
    public static void main(String[] args) {
        B[] b = new B[10];
    }
}

class B{
    static {
        System.out.println("B初始化了");
    }
}
```

如上情况，B类是不会被初始化的

- 引用常量不会触发类的初始化（常量在编译阶段就放入调用类的常量池中了）

深入类加载器

类加载器的层次结构

- 引导类加载器：加载Java的核心包（使用C编写）
- 扩展类加载器：加载Java的扩展库
- 应用程序类加载器：记载Java应用程序的jar包
- 自定义加载器：Java程序员通过继承ClassLoader类的方式实现在即的类加载器

除了引导类加载器，其他的加载器都继承自ClassLoader

java.lang.ClassLoader类的基本职责就是根据一个指定的类名称找到或者生成对应的字节码，然后从字节码中生成一个Class对象。

双亲代理机制

当一个类加载器要加载一个类的，自己先不加载类，而是交给他的父类加载器去加载它，如果他的父类还有父亲，那么继续往上提交，若父亲完不成，则向下传递。

这是一种代理机制，并不是所有的类加载机制都是使用的这种委托机制。

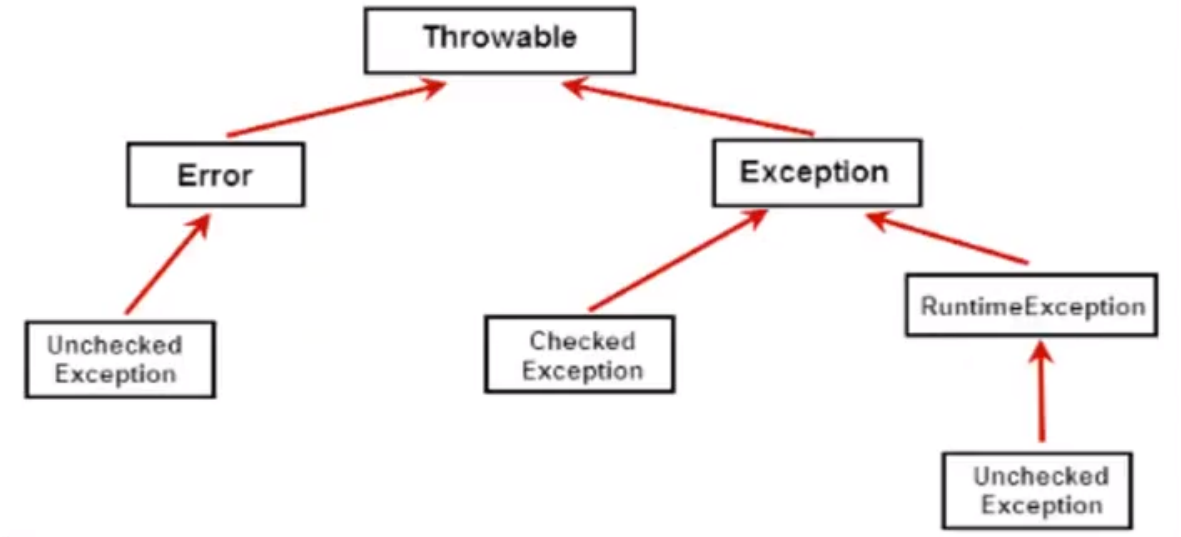
异常

Java采取面向对象的方式来处理异常，处理过程为：

在执行一个方法时，如果发生异常，则这个方法生成一个代表该异常的一个对象，停止当前执行路径，并把异常对象交给运行时环境JRE

运行时环境得到改异常之后，寻找相应的代码来处理该异常，JRE在方法中寻找处理该异常的方法，直到找到相应的处理代码为止。

异常的分类



对于Error类型的错误，表明程序发生了不可逆转的错误，故不需要去管他，应该做的是重启虚拟机。而对于Exception类型的异常，那么就应该引起足够的重视，特别是CheckedException，是我们应该主动去处理的异常。

异常处理

对于运行时异常，一般都是由于程序员的逻辑不严谨所造成的，所以在出现异常的时候应该完善逻辑代码。比如：

```
int a = 0;
if (a != 0) {
    System.out.println(3 / a);
}
System.out.println("hello");
```

对于这种明显异常，我们因该采取逻辑判断及时规避；

而对于编译时异常则需要我们使用对应的try-catch语句或者使用throw方式对其进行处理，编译时异常如果不处理是无法通过编译的，因此必须对其妥善处理。

常见的运行时异常

NullPointerException：空指针异常

通常是一个对象的引用为null的时候却用这个引用去调用了对应对象的属性或者方法。

```
String a = null;
System.out.println(a.length());
```

如上代码便会发生运行时异常。阻止此种异常一般采取的措施为提前判断是否为空

```java
String a = null;
if (a != null){
    System.out.println(a.length());
}
```

ClassCastException：强制类型转换异常

```java
Animal cat = new Cat();
Dog dog = (Dog)cat;
```

一般是将父类类型强制转换为一个具体的子类类型时发生的错误。此种情况下要确定是某种类型的时候才能转型

ArrayIndexOutOfBoundsException：数组越界异常

通常是数组的索引超出了范围

NumberFormatException：格式化数字异常

通常发生在将其他类型转换为数字类型时发生的异常

总结：运行时异常需要我们自己通过逻辑判断去处理

捕获异常（编译器异常）

这种异常是在编译期就要处理的异常，如果不处理，编译无法通过。此时的处理方法一般是通过try-catch-finally方式来处理。

try代码块儿中放可能出现异常的代码

catch代码块放捕获对应异常之后的处理代码

finally代码块不管是否发生异常都会进入这个地方，相当于是对异常处理提供一个统一的出口

```java
        FileReader fileReader = null;
        try {
            fileReader = new
FileReader("I:\\IDEA_project\\abnormal\\src\\main\\resources\\a.txt");
            char read = (char) fileReader.read();
            System.out.println(read);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (fileReader != null){
                    fileReader.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
```

如上就是一个try-catch-finally的例子，注意一点就是子类异常要放在父类异常之前。

抛出异常

此种方式通常是在方法声明上抛出异常，而到时候如果有对象调用了这个方法，那么调用这个方法的对象就应该处理这个异常。

如下代码，我们在readByteByFile方法中将所有的方法都抛出，在main方法中调用类此方法，那么就应该处理抛出此方法的异常。

```
public class ThrowException {

    public static void main(String[] args) {
        char c = 0;
        try {
            c = ThrowException.readByteByFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(c);
    }


    public static char readByteByFile() throws IOException {
        FileReader fileReader = new
FileReader("I:\\IDEA_project\\abnormal\\src\\main\\resources\\a.txt");
        char word = (char) fileReader.read();
        return word;
    }
}
```

通过Throw的方式可以将异常抛给调用者进行处理，如果调用者也不处理那也可以继续往上抛出异常，直到最后抛给JRE，由运行时环境来处理这个异常。

自定义异常

如果要自定义编译时异常，那么应该继承Exception类，如果是自定义运行时异常则应该继承RuntimeException类

如下代码演示自定义运行时异常。

```
public class MyException extends RuntimeException {

    MyException() {
    }

    MyException(String msg) {
        super(msg);
    }

}

class Person {
    String name;
}

class Test1 {
    public static void main(String[] args) {
        Person person = new Person();
        if (person.name == null) {
            throw new MyException("名字不能为空");
```

```
        }
    }
}
```

如下代码演示编译时异常

```
public class MyException extends Exception {

    MyException() {
    }

    MyException(String msg) {
        super(msg);
    }

}

class Person {
    String name;
}

class Test1 {
    public static void main(String[] args) {
        Person person = new Person();
        if (person.name == null) {
            try {
                throw new MyException("名字不能为空");
            } catch (MyException e) {
                e.printStackTrace();
            }
        }
    }
}
```

对于自定义的编译时异常，可以当时就try-catch进行处理，也可以使用向上抛出的方式进行处理，直到交给运行时环境为止。

总结

对于运行时异常，通常是在测试的时候发现了就及时使用逻辑判断对其进行处理，因为编译的时候不会发现潜在的运行时异常，所以要格外小心处理运行时异常可能发生的地方。

而对于编译时异常，在编译的时候就会被检查出来，所以可以使用try-catch和throw的方式进行处理，在日常开发中通常是向上抛出，直到最高级调用者使用try进行处理

装饰器设计模式

```
/**
 * 装饰皮设计模式，coffee模拟
 * 1.抽象组件：需要被装饰的组件的抽象（一般是接口或者抽象父类）
 * 2.具体组件：被装饰的对象
 * 3.抽象装饰类：包含了对抽象组件的引用，以及装饰器共有的方法
 * 4.具体的装饰类：具体实施对组件的装饰
 *
 * @author xiaopu
```

```java
 */
public class Test09 {
    public static void main(String[] args) {
        Coffee coffee = new Coffee();
        System.out.println(coffee.info()+"-->"+coffee.cost());
        Milk milk = new Milk(coffee);
        System.out.println(milk.info()+"-->"+milk.cost());
        Suger suger = new Suger(coffee);
        System.out.println(suger.info()+"-->"+suger.cost());
        Suger suger1 = new Suger(milk);
        System.out.println(suger1.info()+"-->"+suger1.cost());
    }
}


/**
 * 抽象组件
 */
interface Drink {
    /**
     * @return 费用
     */
    double cost();

    /**
     * @return 说明
     */
    String info();
}

/**
 * 具体组件
 */
class Coffee implements Drink {
    private String name = "原味咖啡";

    @Override
    public double cost() {
        return 10;
    }

    @Override
    public String info() {
        return this.name;
    }
}


/**
 * 抽象装饰类
 */
abstract class Decorate implements Drink{

    /**
     * 组件的引用
     */
    private Drink drink = null;

```

```
    Decorate(Drink drink){
        this.drink = drink;
    }

    @Override
    public double cost() {
        return this.drink.cost();
    }

    @Override
    public String info() {
        return this.drink.info();
    }
}


/**
 * 具体装饰类:加牛奶
 */
class Milk extends Decorate {
    Milk(Drink drink) {
        super(drink);
    }

    @Override
    public double cost() {
        return super.cost()*4;
    }

    @Override
    public String info() {
        return super.info()+"加入了牛奶";
    }
}

/**
 * 具体装饰类：加糖
 */
class Suger extends Decorate{

    Suger(Drink drink) {
        super(drink);
    }

    @Override
    public double cost() {
        return super.cost()+20;
    }

    @Override
    public String info() {
        return super.info()+"加入了糖";
    }
}
```