1. JAVA字符串操作

**1.int indexOf(String s)**

该方法用于返回参数字符串s在指定字符串中首次出现的索引位置，当调用字符串的indexOf()方法时，会从当前字符串的开始位置搜索s的位置；如果没有检索到字符串s，该方法返回-1

**2. int lastIndexOf(String str)**

同上，返回的是最后一次出现的索引位置

**3. char charAt(int index)**

使用charAt()方法可将指定索引处的字符返回。

**4. String substring(int begin [ , int end])**

第二个参数可选，截取begin到end处的字符串并将其返回

eg:

String str="Hello World!";

String new\_str=str.substring(2);//llo World!

new\_str=str.substring(2,4);//ll

**5. String trim()**

trim()方法返回字符串的副本，去除字符串前后的空格

eg:

String str=" Hello World! ";

String new\_str=str.trim();//Hello World!

**6. boolean equals()**

  如果两个字符串具有相同的字符和长度，则使用equals()方法比较时，返回true。同时equals()方法比较时区分大小写

**注意：JAVA一定不能用”==”来比较字符串！这样比较的不是字符串而是字符串所在的地址**

**7. String replace(CharSequence target, CharSequence replacement)**

如果replacement来替换掉所有的target子串，因为两个参数的类型声明都是CharSequence接口，这就意味着参数可以是CharBuffer, Segment, String, StringBuffer, StringBuilder等其他实现了CharSequence接口的类

eg:

  String str="Hello World!";

String new\_str=str.replace("o","PEP");

**8. 大小写转换**

这两个方法可以实现字符串中的所有字符的大小写互相转换

eg:

str.toLowerCase();

str.toUpperCase();

**9. split (** **String regex)**

使用split()方法可以使字符串按指定的**分隔字符**或**字符串**或**正则表达式**对内容进行分割，并将分割后的结果存放在字符数组中

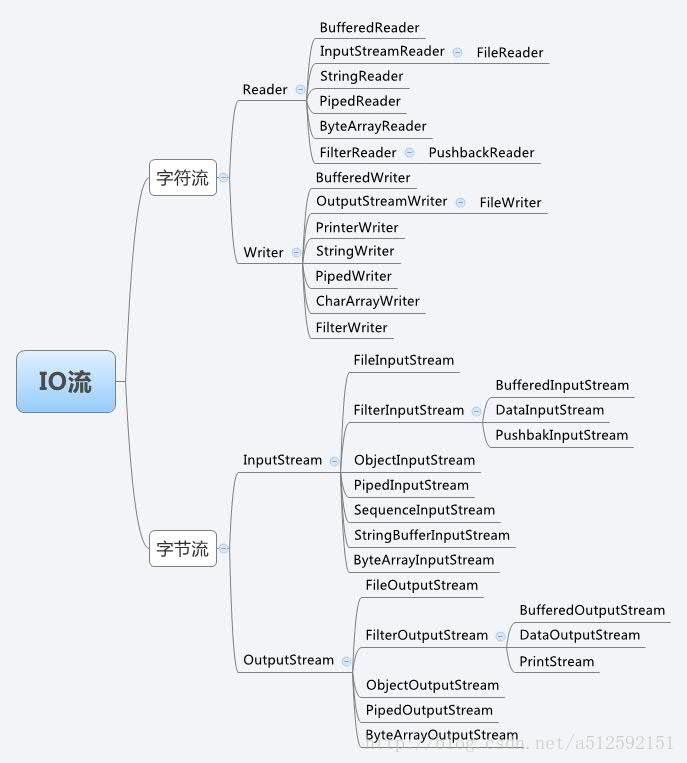
eg:

String str="Hello World!";

String arr[]=str.split("l");

  //["He","","o Wor","d!"]一共四个

1. JAVA 数组操作
2. JAVA I/O流输入输出



* 1. 流的概念

流是取自UNIX的管道(pipe)的概念，用来实现程序或进程之间的通信，宛如流水一般。一个流，必须有源端和目的端，它们可以是内存中的某处地址，也可以是磁盘的某个文件，甚至可以是Internet的某个URL。输入端可以看成数据的生产者，输出端可以看成数据的消费者

根据流的方向，可以将流分为**输入流**和**输出流**，用户从输入流读取信息，将要输出的信息写入输出流

根据流的数据来源，可以将流分为**结点流**和**过滤器**，结点流是直接从指定位置(如文件，内存，网络等)读写数据，而过滤器是以其他输入(结点流、上一个过滤器)为数据来源，将数据过滤处理后以新的流形式提供给用户

根据流处理的数据类型，还可以将流分为**字节流**和**字符流**，JAVA内使用Unicode编码来存储数据到内存中，InputStream和OutStream提供字节流和字符流之间的相互转换，而Reader和Writer提供JAVA内部Unicode编码和其他字符编码之间的转换。字符流以字符为最小单位处理，字节流以字节为最小单位处理

不同编码类型不同字符字符占用的字节大小是不同的，如：

|  |  |  |  |
| --- | --- | --- | --- |
| 字符类型  编码类型 | 汉字 | 数字 | 英文字母 |
| utf-8 | 3~4字节 | 1字节 | 1字节 |
| GBK | 2字节 | 1字节 | 1字节 |
| ASCII | 不支持 | 1字节 | 1字节 |
| JAVA内的Unicode编码 | 2字节 | 2字节 | 2字节 |

**字节流常用于读取各种二进制类型的数据如音乐、视频、图片、word文档(word文档不是文本文件，原因自己查)等**

**字符流常用于读取文本文件等**

* 1. 控制台输入输出

在讨论字节流和字符流之前，需要了解有关控制台的输入输出。与其他流输入输出不同，控制台输入输出是定义在JAVA lang包中的，其他流式输入输出都是定义在JAVA IO包中的。System.in和System.out就是其控制台输入输出的类对象，他们是静态的，也就是说他们不需要我们手动new创建，而是驻留在JVM中，当虚拟机JVM启动时自动创建。他们代表的是标准输入输出。默认情况下，输出设备是显示器，输入设备是键盘。

**1.system.in.read()** 一次读取满一个byte数组(或者读到结束)，然后返回该byte数组

**2.** **int read(byte[] b)**　  //读入多个字节到缓冲数组b中，返回值是读入的字节数

eg：

**try** {

**char** c=(**char**) System.*in*.read();

System.*out*.println(c);

} **catch** (IOException e) {

e.printStackTrace();

}

//输入 shdj 输出 s

//输入 张三 输出 ?

**JAVA util包还提供了一个scanner类供我们读取控制台输入，但该类效率太低，不建议使用**

* 1. 基础知识
     1. 阻塞

由于java采用流式输入输出，因此可能会产生阻塞问题，比如一个输入流读取控制台输入。当运行程序时，程序等待输入，这个等待过程中，程序不再继续执行，而是等待用户输入数据再继续执行，这个过程被称为阻塞

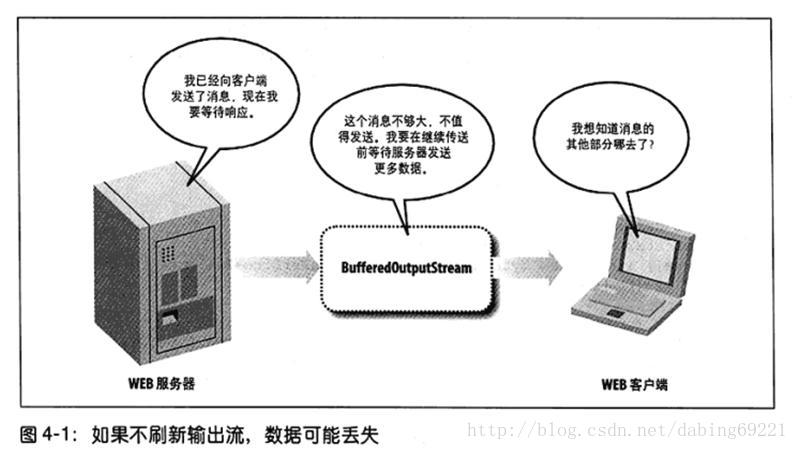
通常的，java IO流中有两种工作方式：**阻塞式**与**非阻塞式** 。

**阻塞方式**：当试图对某个数据源进行读写时, 如果当时数据源没有东西可读,或者目的地暂时不可写, 当前线程上的程序就进入等待状态, 直到有东西可读或者可写为止。这样会造成整个线程阻塞住，如果用户较多，就得一个用户一个线程，极度浪费资源

**非阻塞方式**：如果没有东西可读, 或者不可写, 读写函数马上返回, 而不会等待。可以通过available()方法实现，该方法返回当前流中可读的字节数，通过判断该方法的返回值判断读写是否阻塞

参考：<https://www.cnblogs.com/xiohao/p/5482219.html>

* + 1. flush()刷新输出流



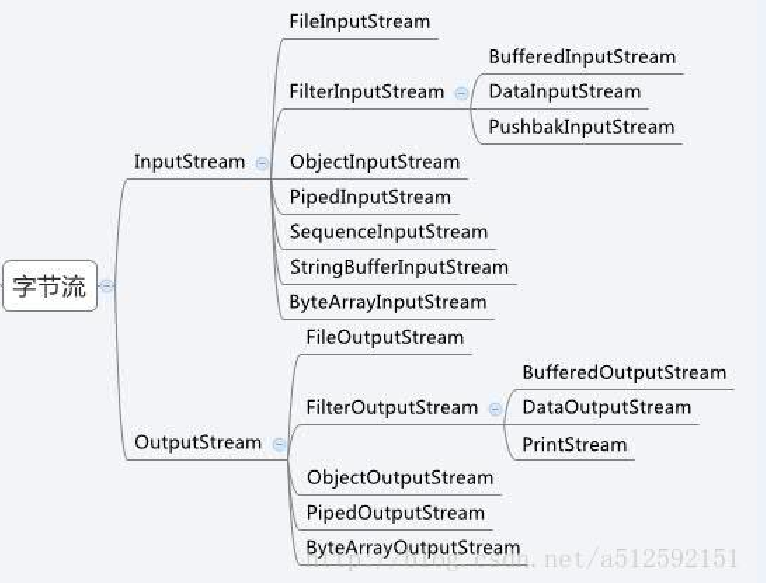
如果我们使用了Buffered缓冲区来装饰我们的输出流类，使我们的输出流具有了缓冲的功能，那么可能会出现如下情况：

服务器通过输出流向客户端响应了一个300字节的信息。但是，这时的输出流有一个1024字节的缓冲区。所以，输出流就一直等着WEB服务器继续向客户端响应信息，只有当WEB服务器的响应信息把输出流中的缓冲区填满时，这时，输出流才向WEB客户端响应消息。当服务器一次发送的数据无法填满缓冲区时，就出现了死锁。

 为了解决这种尴尬的局面，flush()方法出现了。flush()方法可以强迫输出流(或缓冲的流)发送数据，即使此时缓冲区还没有填满，以此来打破这种死锁的状态。如果没有使用flush()就贸然使用close()关闭输出流，那么这部分缓冲区的数据就会丢失，所以，当需要关闭输出流时，请务必flush()一下(当然大部分时候close()方法中包含了flush()，但也有例外，如socke的套接字)

* 1. JAVA字节流

InputStream和OutStream是所有字节输入、输出流的祖先



* + 1. FileStream文件输入输出

字节流文件输入输出主要用于读写二进制文件(图片，音乐，视频等)

* + - 1. File类

File4文件类是java中从文件对象中抽象出的类，它既可以指代具体的某个文件，也可以指代某个目录(文件夹)

可以通过**new File(path)**来创建一个File对象，如果该对象是一个目录，还可以使用**list()**方法获取目录下的所有文件的文件名

**1.list([Filter])** 返回目录下的所有文件的文件名

**注意： (1)可以传递一个可选的参数来过滤文件名**

**(2)File对象的toString()会将File对象转化为文件的绝对路径**

eg：

**try** {

File file=**new** File("E:\\");

String lists[]=file.list();

**for** (String dir\_name: lists) {

System.*out*.println(dir\_name);

}

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**2.listFiles()** 返回目录下的所有文件的File对象

eg：

File file=**new** File("E:\\");

File[] lists=file.listFiles();

**3. createNewFile()** 如果该file文件不存在则创建，创建成功返回 true；如果指定的文件已经存在，则返回 false

eg：

File file=**new** File("E:\\ ");

file.createNewFile();

**4. mkdir ()** 如果该file文件夹不存在则创建，创建成功返回 true；如果指定的文件夹已经存在，则返回 false

eg：

File file=**new** File("E:\\ss");//创建文件ss

file. mkdir();

**5. mkdirs()** 如果该file文件夹不存在则创建，创建成功返回 true；如果指定的文件夹已经存在，则返回 false

eg：

File file=**new** File("E:\\ss");//创建文件夹ss

file.createNewFile();

其他的用到自己查jdk文档

* + - 1. FileInputStream文件输入

**1. public int available()**

该方法返回流中目前实际可读字节数，也就是总大小，这部分字节可以不受阻塞地被读取出来

**2. public void close()**

关闭流

**3. read()**

文件流的read()方法有两种版本，一个是带参数的，一种是不带参数的，这两个方法在读到EOF文件结尾时都返回-1：

**public int read()** 逐字节读，一次从文件中读取一个字节,返回下一个数据字节，遇到EOF返回-1

**public int read(byte[] b)** 以字节数组读取数据，一次从文件中读取b.length个字节到b数组中，返回读入比特数组的字节数，遇到EOF返回-1

* + - 1. FileOutStream文件输出

**1. public int flush()**

刷新缓冲区

**2. public void close()**

关闭流

**3. write()**

write()有三种版本：

**public void write(int b)** 写入单个字节。

**public void write(byte[]  b)** 写入字节数组。

**public void write(byte[]  b,**

**int off,**

**int len)** 写入字节数组的某一部分。从off开始，写len个

**eg：**

**try** {

FileInputStream fin=**new** FileInputStream(**new** File("C:\\Users\\RicardoDing\\Desktop\\20140814122633546.jpg"));

FileOutputStream fout=**new** FileOutputStream(**new** File("C:\\Users\\RicardoDing\\Desktop\\新照片.jpg"));

**byte**[] data=**new** **byte**[1024];

**int** len=0;

**while**((len=fin.read(data))!=-1){

fout.write(data,0,len);

System.*out*.println(len);

}

fin.close();

fout.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

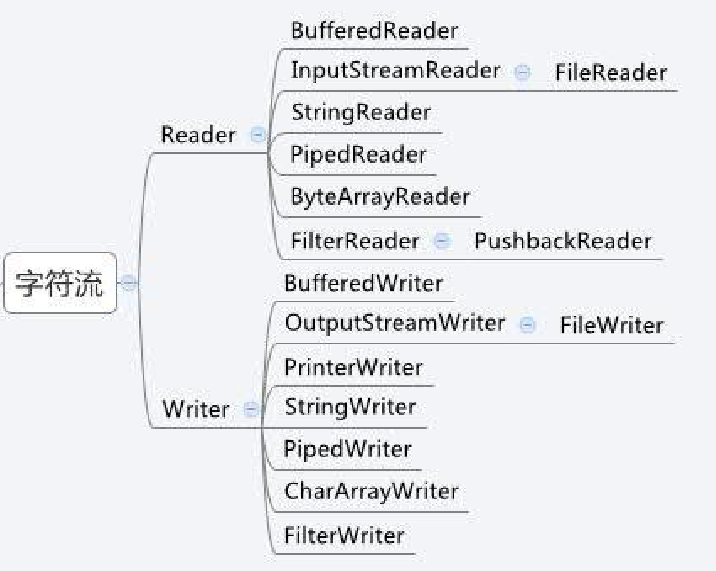
// **TODO** Auto-generated catch block

e.printStackTrace();

}

* 1. JAVA字符流

Reader和Writer是所有字符输入、输出流的祖先



* + 1. File文件输入输出

字符流的文件输入输出一般用于读写文本文件

#### 3.5.1.1 FileReader文件输入

**1. public void close()**

关闭流

**2. read()**

文件流的read()方法有两种版本，一个是带参数的，一种是不带参数的，这两个方法在读到EOF文件结尾时都返回-1：

**public int read()** 逐字符读，一次从文件中读取一个字符,返回读取的字符(int形式，需要强制转换)，遇到EOF返回-1

**public int read(char[] c)** 以字符数组读取数据，一次从文件中读取b.length个字符到c数组中，返回读入字符数组的字符数，遇到EOF返回-1

eg：

FileReader fr=**new** FileReader(myNewFile);

//方式一(不带缓冲数组)：

**int** ch;

**while**((ch= fr.read())!=-1){

System.*out*.println((**char**)ch);

}

//方式二(带缓冲数组)：

**char**[] arr=**new** **char**[100];

**int** arrLength=0;

**while**((arrLength=fr.read(arr))!=-1){

System.*out*.println(**new** String(arr,0,arrLength));

}

fr.close();

#### 3.5.1.2 FileWriter文件输出

需要注意的是，创建FileWriter对象时，在遇到文件不存在的时候会自动创建(FileReader会报错)，**在遇到文件存在的时候会创建一个新的文件来覆盖旧文件,如果希望不采用覆盖方式而是续写旧文件，加入参数true**

FileWriter fw=**new** FileWriter(myNewFile,**true**);

**1.public void flush()**

刷新缓冲区，强制将缓冲区数据输出

**2.write()**

write()有三种版本：

**public void write(int c)** 写入单个字符。

**public void write(char[] cbuf,**

**int off,**

**int len)** 写入字符数组的某一部分。从off开始，写len个

**public void write(String  str,**

**int off,**

**int len)** 写入字符串的某一部分。从off开始，写len个

**eg：**

FileWriter fw=**new** FileWriter(myNewFile);

//方式一

fw.write('H');

//方式二

**char**[] str\_arr={'h','e','l','l','o'};

fw.write(str\_arr);

//方式三

String str="helloworld";

fw.write(str);

fw.flush();

fw.close();

**注意:**

**1.windows中的换行符是\r\n，而其他系统如Linux的换行符是\n，也就是说**

String str="hello\nworld";

fw.write(str);

**这段代码在不同的系统会有不同的结果，当然JAVA为我们提供了一个方法System.*getProperty*("line.separator")来解决这个问题，将这个函数返回值赋值给一个常量，我们就可以跨平台地写入换行符**

**private** **static** **final** String *LINE\_SEPARATOR* =System.*getProperty*("line.separator");

String str="hello" + *LINE\_SEPARATOR* + "world!!!!!!!!!";

fw.write(str);

**2.通过字符缓冲数组读取文件一定要注意，一旦最后一次读取的数据不够填满整个缓冲数组，缓冲数组就会被部分覆盖，eg：**

**char**[] charBuf=**new** **char**[8];

**while**((fr.read(charBuf))!=-1){

fw.write(charBuf);

}

如果倒数第二次次读取的数据为”哈哈哈哈哈哈哈哈”(8个字符)，最后一次读取的数据为”嘻嘻嘻”(3个字符)，那么最后一次循环的charBuf就会变成”嘻嘻嘻哈哈哈哈哈”，因为它每次读取的数据到缓冲数组中都会覆盖上一次的数据，因此上述代码应该写成：

**char**[] charBuf=**new** **char**[24];

**int** len=0;

**while**((len=fr.read(charBuf))!=-1){

fw.write(charBuf,0,len);

}

### 3.5.2 Buffered缓冲流

缓冲流可以使用默认缓冲区，也可以自己指定缓冲区大小：

**BufferedReader(Reader in, int sz)**

BufferedReader reader = **new** BufferedReader( **new** InputStreamReader(System.*in*),1024 );

#### 3.5.2.1 BufferedReader缓冲输入流

**1. public void close()**

关闭流

**2. public int read()**

读取单个字符。以整数形式返回读取的单个字符，如果已到达流末尾，则返回 -1

**3. public int read(char[] cbuf , int off , int len)**

将字符读入到数组的某一个位置(从off到off+len-1)，返回读取的字符数，如果已到达流末尾，则返回 -1

**4. public String readLine()**

读取一个文本行。遇到下列终止字符之一即可认为本行已终止：换行 ('\n')、回车 ('\r') 或回车后直接跟着换行。返回包含该行内容的字符串(不包含任何行终止符)，如果已到达流末尾，则返回 null

#### 3.5.2.2 BufferedWriter缓冲输出流

**1. public void close()**

关闭流

**2. public void flush()**

刷新缓冲区

**3. public void newLine()**

向输出流中写入一个行分隔符，windows平台下为\r\n，Linux下为\n

**4.write()**

有三种版本：

**public void write(int c)** 写入单个字符。

**public void write(char[] cbuf,**

**int off,**

**int len)** 写入字符数组的某一部分。从off开始，写len个

**public void write(String  str,**

**int off,**

**int len)** 写入字符串的某一部分。从off开始，写len个

* 1. 装饰者模式

装饰着模式和普通的类继承一样，都是为了实现功能的扩展，但装饰着模式可以避免过多过杂的继承，比如，对于File流类和object流类一共四个类(输入输出各俩)进行功能扩展，扩展的功能有缓冲功能和数据输入输出两个功能，如果用继承，就会产生8个类，并且随着功能越来越多，数量也越来越夸张，整个体系越来越臃肿。如果用装饰着模式，只要4个类+2个功能类就可以了，每多一个功能，只需要再加一个装饰类即可，但要注意的是：装饰类和被装饰类都必须是同一个父类的子孙类

* 1. 网络IO
     1. socket套接字编程

要想客户端和服务器能在网络中通信，那必须得使用 Socket 编程，它是进程间通信里比较特别的方式，特别之处在于它是可以跨主机间通信。

Socket的中文名叫作插口，双方要进行网络通信前，各自得创建一个 Socket，这相当于客户端和服务器都开了一个“口子”，双方读取和发送数据的时候，都通过这个“口子”。这样一看，是不是觉得很像弄了一根网线，一头插在客户端，一头插在服务端，然后进行通信。

如果我们要进行socket通讯，需要怎么做？

首先服务端需要创建一个ServerSocket()对象，在它的构造函数中，将调用bind()方法将ServerSocket和IP地址、端口号、以及请求连接队列的最大长度backlog进行绑定，为什么要绑定IP和端口呢？

* 1. **绑定端口的目的：**当内核收到 TCP 报文，通过 TCP 头里面的端口号，来找到我们的应用程序，然后把数据传递给我们。
  2. **绑定 IP 地址的目的：**一台服务器是可以有多个网卡的，每个网卡都有对应的 IP 地址，当绑定一个网卡时，内核在收到该网卡上的包，才会发给我们；

接着bind()方法将调用listen()方法对连接进行监听，此时构造函数创建ServerSocket()对象完成

服务端进入了监听状态后，通过调用 accept() 函数，来从内核获取客户端的连接，如果没有客户端连接，则会阻塞等待客户端发起连接

接着客户端创建Socket，指明要连接的服务端的IP和端口，在socket()的构造函数中，将调用connect()方法发起连接，接下来就是我们熟悉的TCP三次握手阶段，代码如下：

服务端代码:

public class SocketServer {  
 public static void main(String[] args) {  
 try {  
 // 创建服务端socket  
 ServerSocket serverSocket = new ServerSocket(8088);  
 // 创建客户端socket  
 Socket socket = new Socket();  
  
 //循环监听等待客户端的连接  
 while(true){  
 // 监听客户端  
 socket = serverSocket.accept();  
 //使用自定义线程ServerThread，处理服务端收到的数据

ServerThread thread = new ServerThread(socket);  
 thread.start();  
 InetAddress address = socket.getInetAddress();  
 System.*out*.println("当前客户端的IP："+address.getHostAddress());  
 }  
 } catch (Exception e) {  
 // *TODO: handle exception* e.printStackTrace();  
 }  
 }  
}

ServerThread代码：

public class ServerThread extends Thread{  
 private Socket socket = null;  
 public ServerThread(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 InputStream is=null;  
 InputStreamReader isr=null;  
 BufferedReader br=null;  
 OutputStream os=null;  
 PrintWriter pw=null;  
 try {  
 is = socket.getInputStream();  
 isr = new InputStreamReader(is);  
 br = new BufferedReader(isr);  
 String info = null;  
 while((info=br.readLine())!=null){  
 System.*out*.println("我是服务器，客户端说："+info);  
 }  
 socket.shutdownInput();  
 os = socket.getOutputStream();  
 pw = new PrintWriter(os);  
 pw.write("服务器欢迎你");  
 pw.flush();  
 } catch (Exception e) {  
 // *TODO: handle exception* } finally{  
 //关闭资源  
 try {  
 if(pw!=null)  
 pw.close();  
 if(os!=null)  
 os.close();  
 if(br!=null)  
 br.close();  
 if(isr!=null)  
 isr.close();  
 if(is!=null)  
 is.close();  
 if(socket!=null)  
 socket.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

客户端代码：

public class SocketClient {  
 public static void main(String[] args) throws InterruptedException {  
 try {  
 // 和服务器创建连接  
 Socket socket = new Socket("localhost",8088);  
  
 // 要发送给服务器的信息  
 OutputStream os = socket.getOutputStream();  
 PrintWriter pw = new PrintWriter(os);  
 pw.write("客户端发送信息");  
 pw.flush();  
  
 socket.shutdownOutput();  
  
 // 从服务器接收的信息  
 InputStream is = socket.getInputStream();  
 BufferedReader br = new BufferedReader(new InputStreamReader(is));  
 String info = null;  
 while((info = br.readLine())!=null){  
 System.*out*.println("我是客户端，服务器返回信息："+info);  
 }  
  
 br.close();  
 is.close();  
 os.close();  
 pw.close();  
 socket.close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

TCP Socket是最简单、最基本的网络通信，它基本只能一对一通信，因为使用的是同步阻塞的方式，当服务端在还没处理完一个客户端的网络 I/O 时，或者 读写操作发生阻塞时，其他客户端是无法与服务端连接的。

但是在现代网络编程中，这种一对一模式的IO模型并不能满足我们的要求，为了实现服务器对多客户端的支持，逐渐发展出一系列的IO模型

* + 1. IO的两个阶段——等待就绪和操作
       1. 用户态和内核态

当我们需要从磁盘读取某个文件时，并不是直接加载文件到应用程序CPU内存中的，而是：

1. 先将数据从**磁盘**复制到**内核Buffer**中
2. 再将数据从**内核Buffer**复制到**用户Buffer**中

CPU指令根据其重要程度，分为不同权限，有的指令执行失败会导致整个系统崩溃，如果将这些指令**随意**开放给应用程序，则会加大系统崩溃的概率，因此我们将CPU指令分为内核态指令和用户态指令。当我们需要进行系统调用或者进行中断时，我们就需要将我们的CPU核心从用户态切换到内核态

这就好像我们去行政大厅办事一样，用户提出要办的事项，由公务员帮助我们办理(系统调用)，如果我们给普通市民放开政府系统的权限，就容易出现各种问题

用户态切换到内核态有3种方式：

1. 系统调用：用户态线程希望执行高级别的CPU指令，主动要求切换到内核态
2. 中断：比如硬盘读写完成后，系统会中断当前cpu的程序转而执行磁盘读写的中断处理程序(要进入内核态)，如果被中断的线程处于用户态，此时会发生切换
3. 异常：当CPU执行用户态下的指令时，如果发生了某些异常，就会触发处理此异常相关的程序(要进入内核态)，比如缺页异常
   * + 1. 两个阶段—等待就绪和操作

我们网络IO分为两个阶段：

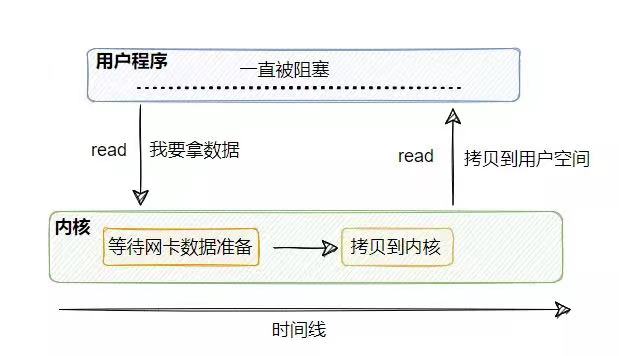
1. **等待就绪：**这个阶段客户端可能还有部分数据正在发送，也就是说我们内核Buffer中的数据不完整。程序需要等待数据从网卡拷贝到内核buffer中
2. **操作：**当内核态的数据已经接收完毕，就会将内核态Buffer中的数据复制到用户态Buffer中，也就是我们真正的进行IO操作，这样我们处于用户空间的应用程序才能访问这些数据
   * 1. 五种IO模型

我们常用的IO模型主要有五种：

1. BIO
2. NIO
3. IO多路复用
4. 信号驱动IO
5. 异步IO
   * + 1. BIO(Blocking I/O)同步阻塞IO

**在等待就绪阶段：**我们的BIO发起read系统调用，CPU从用户态切换到内核态，由于内核Buffer中的数据还未Ready，线程会阻塞

**在操作阶段：**内核就绪后，会将内核Buffer的数据复制到用户Buffer中，这一阶段仍是阻塞的，直到复制结束，IO完成



传统的阻塞式IO是为每一个连接分配一个线程，伪代码如下：

{  
 ExecutorService executor = Excutors.newFixedThreadPollExecutor(100);//线程池  
 ServerSocket serverSocket = new ServerSocket();  
 serverSocket.bind(8088);  
 while(!Thread.currentThread.isInturrupted()){//主线程死循环等待新连接到来  
 Socket socket = serverSocket.accept();  
 executor.submit(new ConnectIOnHandler(socket));//为新的连接创建新的线程  
 }

}  
  
class ConnectIOnHandler extends Thread{  
 private Socket socket;  
 public ConnectIOnHandler(Socket socket){  
 this.socket = socket;  
 }  
 public void run(){  
 while(!Thread.currentThread.isInturrupted()&&!socket.isClosed()){

//死循环处理读写事件  
 String someThing = socket.read()....//读取数据  
 if(someThing!=null){  
 ......//处理数据  
 socket.write()....//写数据  
 }  
  
 }  
 }  
}

socket.accept()、socket.read()、socket.write()三个主要函数都是同步阻塞的。

如果采用单线程进行BIO，一旦阻塞，单线程就会停滞，此时不具备任何并发能力。

如果采用多线程进行BIO (一般还会采用线程池降低频繁创建和销毁线程的开销)，虽然具备了并发能力，但依然存在很多缺点

在连接数不是很多时，这种模式没有什么问题

优点：

**1.利用多核。**

**2.当I/O阻塞系统，但CPU空闲的时候，可以利用多线程使用CPU资源。**

缺点：

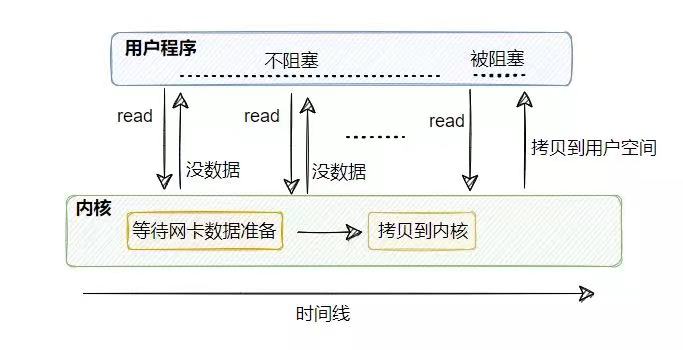
1. **当连接数过多时，需要建立大量线程，而线程建立是有内存开销的，由于IO阻塞，大量线程实际上创建了以后一直处于阻塞状态，只有少量线程在执行，这部分阻塞的线程无法处理其他客户端的IO请求，严重浪费资内存源**
2. **建立线程过多，上下文切换开销严重**

面对单机1000以下的连接，BIO可以胜任，但遇上十万甚至百万级连接的时候，传统的BIO模型是无能为力的，此时就需要一种更高效的IO方式

* + - 1. NIO(non Blocking I/O)非阻塞式IO

**在等待就绪阶段：**我们的BIO发起read系统调用，CPU从用户态切换到内核态，由于内核Buffer中的数据还未Ready，内核会立即返回一个错误，随后用户线程不断发起请求，直到数据就绪

**在操作阶段：**内核就绪后，会将内核Buffer的数据复制到用户Buffer中，**这一阶段仍是阻塞的，直到复制结束**，IO完成



NIO是非阻塞式的，如果有数据收到，就返回数据，如果没有数据收到，就立刻返回错误。也就是说，程序会不停地去尝试读取数据，但是不会阻塞在那个读方法里，如果读的时候，没有读到内容，也会立即返回。这就允许我们在程序里，在读到不数据的时候可以去做别的事情，而不是一直阻塞在那里。

以外卖举例，就相当于，我一边扫地，一边等外卖。不再像原来一样，在门口傻等了，而是扫两下，就跑到门口看看外卖到了没有。一直这样循环，直到取到外卖，才从这个循环中跳出来，进入吃的流程。

这种模式最大的问题在于一直轮询所有的连接，包括没有发生读写的连接，这会很浪费 CPU，导致cpu空转。**我们通常把这种模式叫做忙轮询。后面还会介绍无差别轮询(select/poll)和epoll**

NIO不仅有CPU空转的问题，为每一个客户端请求建立一个线程专门处理，依然会占用大量的虚拟机资源。那么，是否可以让单个线程来监听多个客户端呢？

我们可以每 accept 一个客户端连接后，将这个文件描述符client\_fd放到一个数组list里，然后无限轮询这个list。这样，我们就成功用一个线程处理了多个客户端连接，伪代码如下：

// 伪代码描述  
while(true){  
 // accept非阻塞（cpu一直忙轮询）  
 client\_fd = accept(listen\_fd)  
 if(client\_fd != null){  
 // 有人连接,连接成功，加入到集合中  
 list.add(client\_fd)  
 }else{  
 // 无人连接  
 }  
 for(fd in list){  
 // 设置连接为非阻塞  
 setNonblocking(client\_fd)  
 // read 为非阻塞命令  
 if(len = read(fd) && len > 0){  
 // 有读写数据，进行数据处理  
 }else{  
 // 无读写数据，轮询下一个连接  
 }  
 }  
}

看似我们解决了线程资源占用过多的问题，但这仍然不划算，这个操作就好像我们在循环里调用RPC，在循环里一条一条地插入sql一样。因为read系统调用仍然是一个非常重的操作，每次read都设计内核态和用户态的转换。为此，我们不得不求助于操作系统，由操作系统提供给我们一个有这样效果的函数(select)，我们将一批文件描述符通过一次系统调用传给内核，由内核层去遍历，才能真正解决这个问题

* + - 1. IO多路复用

在之前的BIO和NIO中，只涉及到一种系统调用，即read()，IO多路复用引入了新的系统调用SELECT

**在等待就绪阶段：**我们的BIO发起select系统调用，CPU从用户态切换到内核态，查看多个socket的状态

如果没有就绪状态的socket，就阻塞调用select的线程，防止CPU空转，同时内核会监控所有select负责的socket，一旦某个socket就绪，select就会返回

如果有就绪状态的socket，就返回可操作的socket

**在操作阶段：**内核就绪后，会将内核Buffer的数据复制到用户Buffer中，这一阶段仍是阻塞的，直到复制结束，IO完成

I/O多路复用可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。简而言之，就是可以用一个线程同时监控多个描述符是否就绪

* + - * 1. SELECT

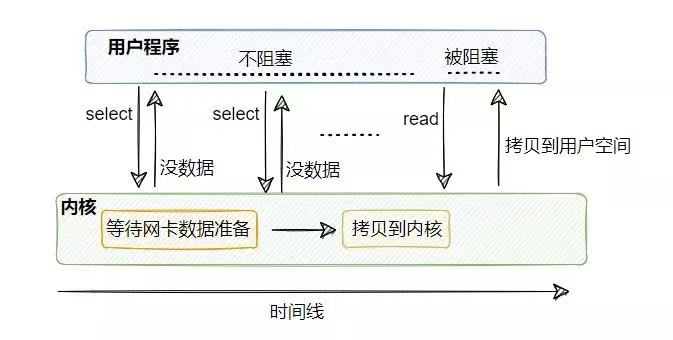
为了解决CPU空转的问题，我们引入了一个代理select，这个代理可以同时观察许多流的I/O事件，在空闲的时候(即没有流处于读/写就绪状态)，会把当前线程阻塞掉，防止不断轮询CPU空转。然后SELECT代理继续观察所有流，一旦有一个或多个流有I/O事件时，就从阻塞态中醒来，处理数据，于是我们的程序就会轮询一遍**所有的流**。代码长这样:

// 伪代码描述  
while(true) {  
 // 观察所有的连接，如果没有任何一个连接处于receive就绪状态，则select阻塞，让出CPU，直到有一个或多个连接处于IO就绪状态后，唤醒本线程继续执行

fds = select(fds)  
 for (fd in fds) {  
 if (len = receive(fd) && len != -1) {  
 // 处理数据  
 }

}  
}

select既做到了一个线程处理多个客户端连接（文件描述符），又减少了系统调用的开销，大幅提高了IO效率



那么select是如何实现同时观察多个socket的呢？

select 实现多路复用的方式是：将已连接的 Socket 都放到一个文件描述符集合，然后调用 select 函数将文件描述符集合**拷贝**到内核里，让内核来检查是否有网络事件产生。检查的方式也很粗暴，就是通过**遍历**文件描述符集合的方式，当检查到有事件产生后，将此 Socket 标记为可读或可写。 接着再把整个文件描述符集合拷贝回用户态里，然后用户态**还需要再通过遍历的方法找到可读或可写的 Socket**，然后再对其处理，并且， select支持的socket数量是**有上限的**，默认最大1024个。

从上面加粗的字体中，我们可以看到select的缺陷，**它至少需要两次遍历socket集合**，第一次在内核里，第二次在用户态中。另外，**它还要对socket集合进行两次拷贝**，先从用户空间拷贝到内核空间，对就绪的socket做完标记后再从内核空间拷贝到用户空间。

另外，select 使用固定长度的 BitsMap，表示文件描述符集合，而且所支持的文件描述符的个数是有限制的，在 Linux 系统中，由内核中的 FD\_SETSIZE 限制， 默认最大值为 1024，只能监听 0~1023 的文件描述符，为了解决这个限制，Poll出现了

* + - * 1. POLL

poll本质上和select没有区别，它将用户传入的socket数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是poll 不再用 BitsMap 来存储文件描述符，取而代之用链表形式来组织，突破了 select 的文件描述符个数限制，当然还会受到系统文件描述符总数限制。

但是 poll 和 select 并没有太大的本质区别，都是使用「线性结构」存储进程关注的 Socket 集合，因此都需要遍历文件描述符集合来找到可读或可写的 Socket，时间复杂度为 O(n)，而且也需要在用户态与内核态之间拷贝文件描述符集合，这种方式随着并发数上来，性能的损耗会呈指数级增长。

* + - * 1. EPOLL

SELECT存在的问题

1. select的连接数是有限制的，最大可以支持1024个文件描述符，epoll去掉了这个限制
2. 每次调用select，都需把文件描述符fd集合从用户态拷贝到内核态，fd很多时开销就很大(epoll优化为在内核中保存一份文件描述集合，不用每次都复制)

具体，操作系统提供了这三个函数：

1. 创建一个 epoll 句柄

int epoll\_create(int size);

1. 向内核添加、修改或删除要监控的文件描述符。

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

1. 类似发起了 select() 调用，等待被管理的文件描述符就绪

int epoll\_wait(int epfd, struct epoll\_event \*events, int max events, int timeout);

1. select仅仅知道有IO事件发生，并不知道具体哪个流就绪了，因此只能由用户进程无差别轮询所有流，找出能够读/写的流，然后进入操作阶段，将数据从内核态Buffer复制到用户态Buffer。所以其事件复杂度为O(n)，同时处理的流越多，每一次无差别轮询时间就越长(epoll优化为只返回给用户就绪的文件描述符，无需用户进程再做无效的遍历)

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll之会把哪个流发生了怎样的I/O事件通知我们。此时我们对这些流的操作都是有意义的，复杂度降低到了O(1)

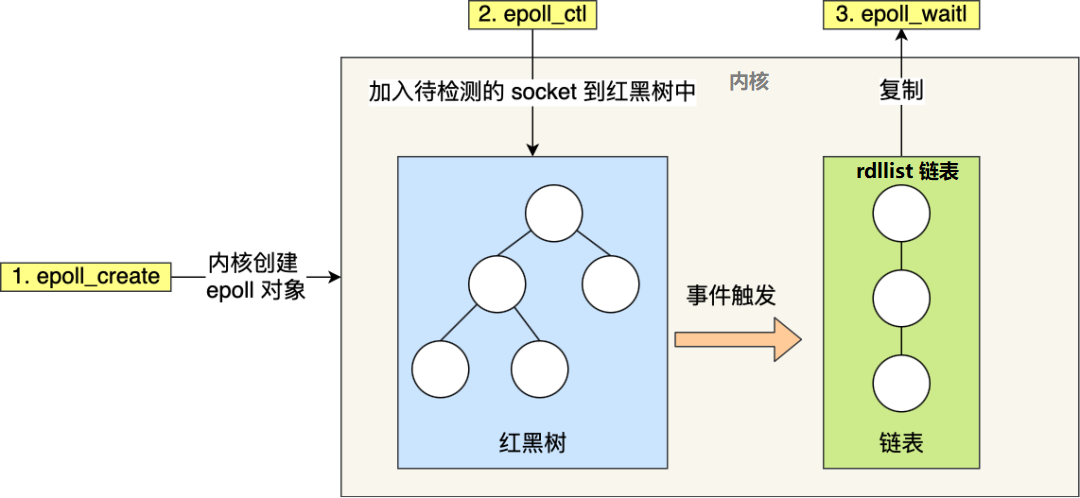
1. 如果采用主动轮询的方式查看哪个流就绪，即使是在操作系统内核中遍历，依然很消耗资源，只不过省去了内核态用户态切换的开销(epoll修改为异步通知，由就绪的流主动通知用户进程)

每当有的文件描述符就绪的时候，内核会主动把就绪的连接放到 rdllist 链表里。应用进程只需要调用**epoll\_wait()**方法判断rdllist就绪链表是否为空即可，而不用去遍历整棵树(文件描述符以红黑树形式组织保存在内核中)。

如果没有IO事件，epoll也会阻塞当前进程

这样就避免了主动轮询整个socket列表

epoll的具体原理如下图：



边缘触发（edge-triggered，ET）

使用边缘触发模式时，当被监控的 Socket 描述符上有可读事件发生时，epoll\_wait会将就绪的描述符从内核态拷贝到用户态，下一次再调用epoll\_wait时，即使用户进程没有调用 read 函数从内核读取完数据，epoll\_wait也不会返回重复的socket描述符，即每个socket描述符每次就绪最多只返回一次；

* 1. 对于读事件：每次被通知读的时候，一定要把socket中的数据读完，为此，我们会循环从文件描述符读写数据，那么如果文件描述符是阻塞的，没有数据可读写时，进程会阻塞在读写函数那里，程序就没办法继续往下执行。所以，边缘触发模式一般和非阻塞 I/O (没有数据时返回错误而不是阻塞)搭配使用，程序会一直执行 I/O 操作，直到系统调用（如 read 和 write）返回错误
  2. 对于写事件：若发送缓冲区不满，epoll会通知一次write事件，直到开发者填满缓冲区后，epoll才会在下一次缓冲区由满变为不满的时候再次通知

1. 优点：降低了epoll\_wait由阻塞状态苏醒过来的概率，减少了内核态和用户态上下文切换的次数，因此效率较高
2. 缺点：对用户程序编程提出了要求，必须使用非阻塞读写

如果处于性能考虑，推荐使用ET模式+非阻塞读写

水平触发（level-triggered，LT，默认）

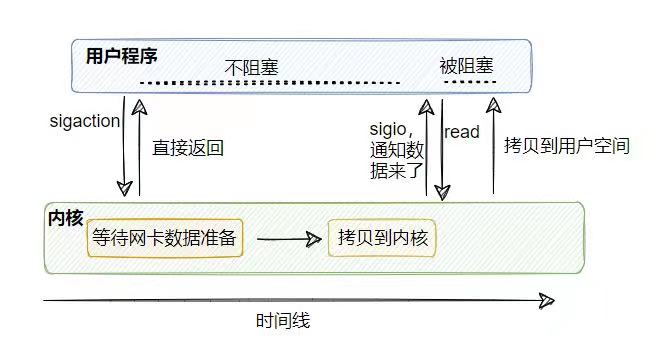
使用水平触发模式时，当被监控的 Socket 上有可读事件发生时，服务器端不断地从 epoll\_wait 中苏醒，直到内核缓冲区数据被 read 函数读完才结束；即只要没有读取完数据，epoll\_wait就会一直返回该socket

select和poll只有LT水平触发模式，epoll默认是水平触发模式

* 1. 对于读事件： LT模式的读事件很好处理，epoll\_wait返回了就读，读多读少都没关系
  2. 对于写事件： LT模式下，如果socket处于空闲状态，一定要及时移出epoll列表中，否则，由于socket空闲相当于一直可写，那么就会一直出现写事件

1. 优点：保证了数据的完整输出
2. 缺点：当单次就绪中，数据量较大时，需要不断调用epoll\_wait，造成大量的用户态内核态之间的上下文切换，影响性能
   * + 1. 信号驱动IO

信号驱动IO是采用sigaction函数向内核注册一个信号函数，然后用户进程继续执行，等到内核数据准备就绪后，就会发送一个sigio信号给用户线程，用户进程在接收到信号后，调用IO读写函数进行IO 请求操作。即注册一个处理动作，然后由内核告知用户线程数据已就绪，再由用户线程去read(还是会阻塞)



信号驱动IO只能用于UDP，而不能用于TCP，这是因为TCP能够产生信号的事件有7种，以下条件均会导致对一个TCP套接字产生SIGIO信号：

* + 1. 监听套接字上某个连接请求已经完成；
    2. 某个断连请求已经发起；
    3. 某个断连请求已经完成；
    4. 某个连接之半已经关闭；
    5. 数据到达套接字；
    6. 数据已经从套接字发送走；
    7. 发生某个异步错误。

这么多条件都会触发SIGIO信号，导致应用进程对该信号一头雾水，没法确定套接字具体发生了什么事情。

而对于UDP，在UDP套接字中，只有以下两个条件会产生SIGIO信号：

1. 数据报到达套接字；
2. 套接字上发生异步错误。

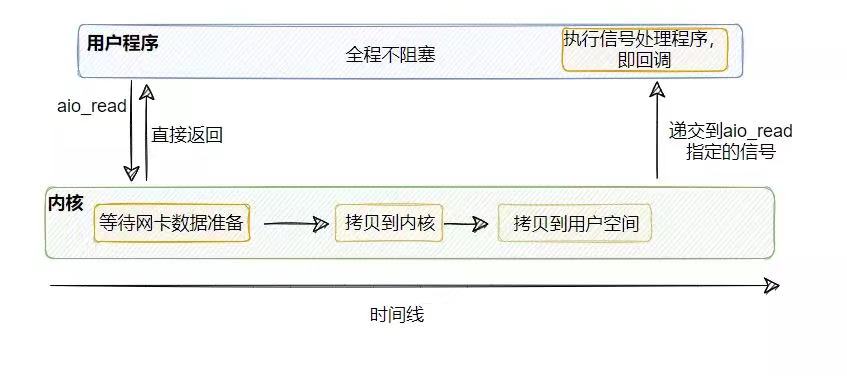
因此信号驱动IO使用较少

* + - 1. 异步AIO

前面的几种模型都是同步IO，因为他们在操作阶段都是需要调用read，阻塞用户线程进行数据读取的。信号驱动IO虽然对TCP不太友好，但其发展思路是对的，即向着异步发展，但它并没有做到异步

**在等待就绪阶段：**我们的BIO发起aio\_read系统调用，CPU从用户态切换到内核态，无论内核Buffer的数据是否就绪，都不会阻塞，立即返回

**在操作阶段：**内核就绪后，会将数据复制到用户态Buffer中，复制完成以后，发送一个Signal给用户线程，或者回调用户线程注册的接口进行通知，用户线程收到通知后读取用户态Buffer中的数据



在异步IO的整个过程种，用户进程不会发生阻塞，做到了真正的异步非阻塞IO

既然异步IO那么好，那么为什么最常用的仍然是IO多路复用，而不是AIO呢？

这是因为在 Linux 系统下，异步IO模型在2.6版本才引入，目前并不完善。所以，这也是在 Linux 下，实现高并发网络编程时都是以 IO 复用模型模式为主。

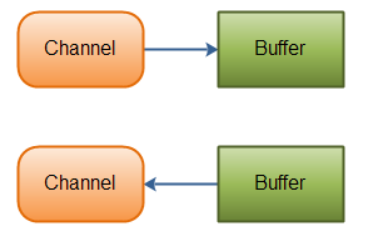
虽然Windows 系统下通过 IOCP 实现了真正的异步 I/O。但是，就目前的业界形式来说，Windows 系统，很少作为百万级以上或者说高并发应用的服务器操作系统来使用，因此主流的IO模型仍然是IO多路复用

* + - 1. BIO\NIO\AIO的区别

区别如下：

1. **BIO是同步阻塞IO，NIO是同步非阻塞的IO，AIO是异步IO非阻塞IO**
2. **BIO面向字节(字符)流，而NIO面向通道和缓冲区**

BIO 以流的方式处理数据，而NIO 以块的方式处理数据，块 I/O 的效率比流 I/O 高很多。在NIO中，数据总是从通道中读到buffer缓冲区内，或者从buffer写入到通道中



BIO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，因为它们没有被缓存在任何地方。所以，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区

NIO面向缓冲区意味着数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。

1. BIO是单向的，要么是输入流要么是输出流，NIO是双向的，在从channel往buffer读写数据的同时，buffer也可以向channel读写数据
   * 1. Reactor模式

由于Linux对AIO的支持还不完善，现有的开源框架大多是基于IO多路复用实现高性能IO的，但IO多路复用是以面向过程的方式编码的，其开发效率较低。因此，基于IO多路复用封装的Reactor模式出现了

Reactor 模式也叫 Dispatcher 模式，即采用 I/O 多路复用监听事件，收到事件后，根据事件类型分配（Dispatch）给某个进程/线程

Reactor模式有两个关键组成：

* + 1. **Reactor：**Reactor运行在一个单独的线程中，负责监听和分发事件，事件类型包含连接事件、读写事件，它就像上世纪的电话接线员一样，负责监听用户的电话并将其分发给适当的联系人；
    2. **Handlers：**处理资源线程池，即IO处理线程，负责处理实际业务逻辑，响应处理IO事件

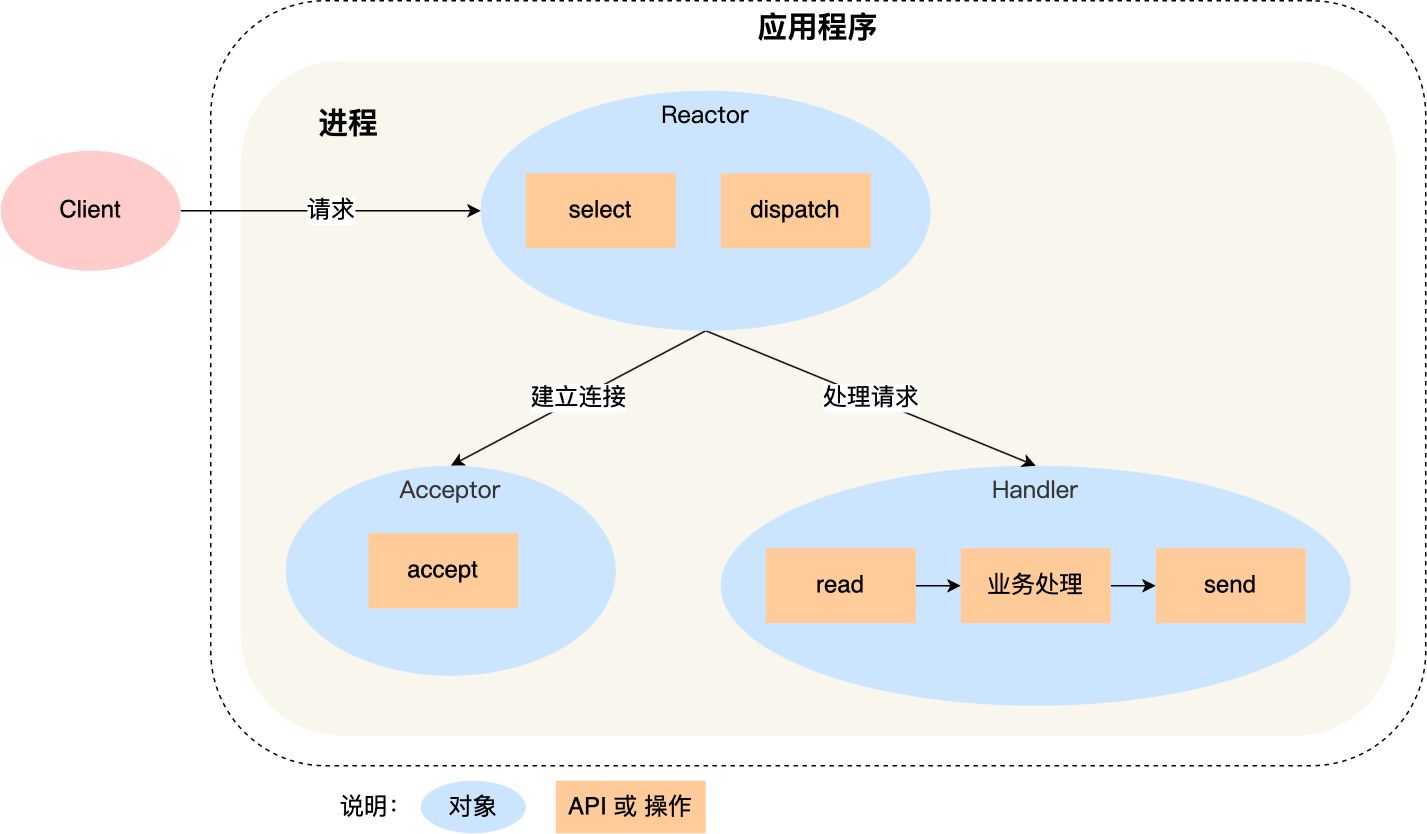
根据Reactor线程的数量和Handlers线程的数量不同，Reactor模式理论上有4种组合，即：

1. 单 Reactor，单进程/线程；
2. 单 Reactor，多进程/线程；
3. 多 Reactor，单进程/线程；
4. 多 Reactor，多进程/线程；

在实际使用中，第三种”多Reactor，单进程/线程”模式，不仅方案更复杂，而且也没有性能优势，因此几乎没有采用该方案的案例

具体实现中采用线程还是进程，则要看使用的编程语言和平台，JAVA语言一般使用的是线程，比如Netty，C语言进程和线程都可以，比如Nginx采用进程，Memcache采用线程

* + - 1. 单 Reactor，单进程/线程



上图中我们可以看到，该模型主要涉及3个组件：

1. Reactor 对象的作用是监听和分发事件；
2. Acceptor 对象的作用是获取连接；
3. Handler 对象的作用是处理业务；

另外的 select、accept、read、send 是系统调用函数，其调用涉及到上下文切换，dispatch 是分发事件的操作

主要流程如下：

* + 1. Reactor 对象通过 select（IO 多路复用接口）监听事件，收到事件后通过 dispatch 进行分发，具体分发给 Acceptor 对象还是 Handler 对象，还要看收到的事件类型；
    2. 如果是连接建立的事件，则交由 Acceptor 对象进行处理，Acceptor 对象会通过 accept 方法获取连接，并创建一个 Handler 对象来处理后续的响应事件；
    3. 如果不是连接建立事件， 则交由当前连接对应的 Handler 对象来进行响应；Handler 对象通过 read -> 业务处理 -> send 的流程来完成完整的业务流程。

优点：

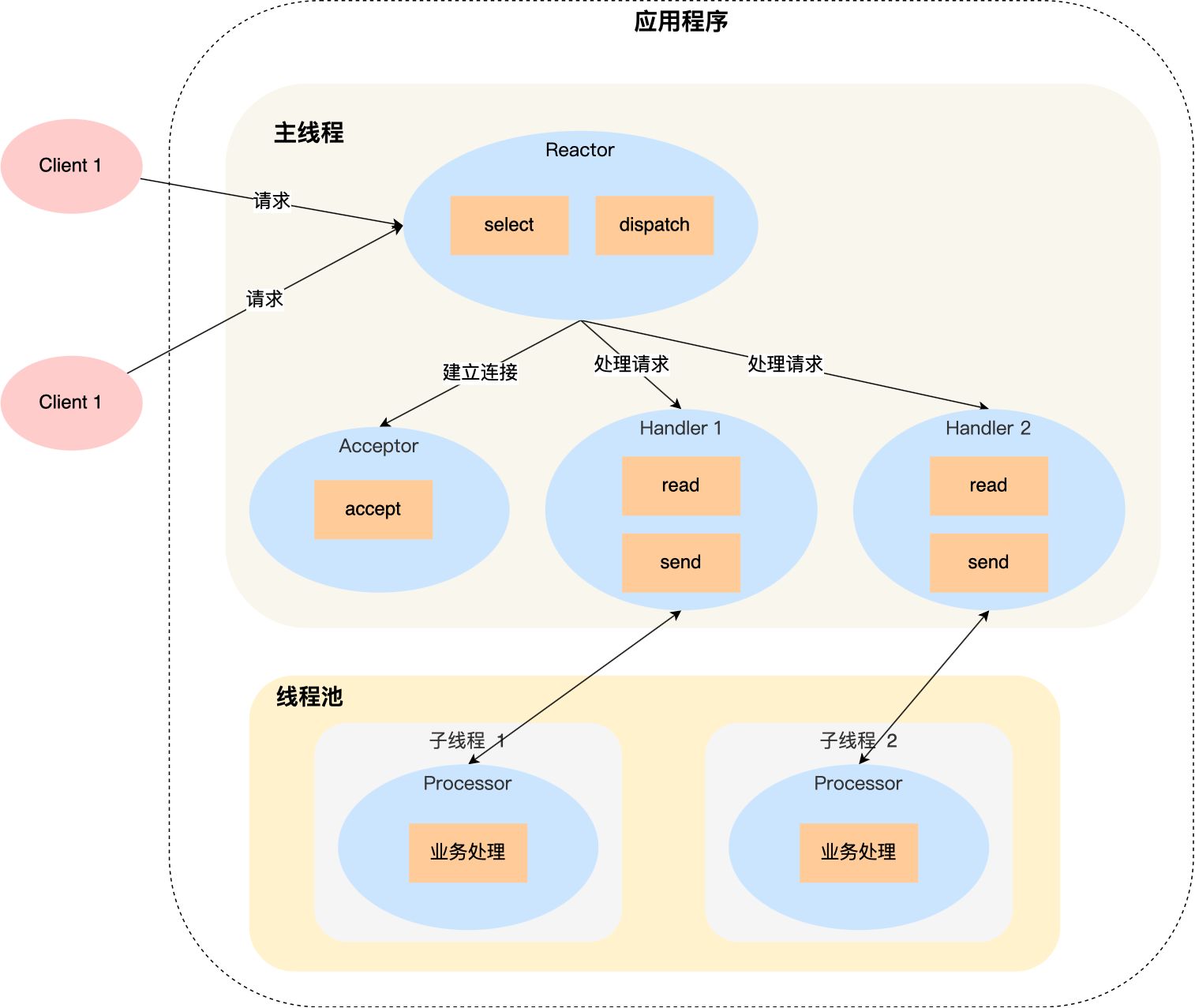
1. 模型简单，Reactor进程/线程负责监听，IO事件由单个线程/进程处理，没有多线程竞争问题

缺点：

1. 性能问题严重，IO事件交由单线程/进程处理，无法发挥多核CPU性能，线程执行某个连接对应的Handler对象时，若其他连接产生IO事件，则无法同步处理
2. 可靠性问题：如果线程/进程进入死循环，或执行时间过长，则会导致整个Reactor模式不可用

因此该模式并不适用于CPU密集型应用，只适用于业务处理非常快的场景，如Redis就是采用该模式的，这是因为redis业务操作大多是在内存中完成，并且采用了高效的数据结构，并不依赖于CPU

* + - 1. 单 Reactor，多进程/线程



该方案的主要流程如下：

1. Reactor 对象通过 select（IO 多路复用接口）监听事件，收到事件后通过 dispatch 进行分发，具体分发给 Acceptor 对象还是 Handler 对象，还要看收到的事件类型；
2. 如果是连接建立的事件，则交由 Acceptor 对象进行处理，Acceptor 对象会通过 accept 方法 获取连接，并创建一个 Handler 对象来处理后续的响应事件；
3. 如果不是连接建立事件， 则交由当前连接对应的 Handler 对象来进行响应；
4. Handler 对象不再负责业务处理，只负责数据的接收和发送，Handler 对象通过 read 读取到数据后，会将数据发给子线程里的 Processor 对象进行业务处理；
5. 子线程里的 Processor 对象就进行业务处理，处理完后，将结果发给主线程中的 Handler 对象，接着由 Handler 通过 send 方法将响应结果发送给 client；

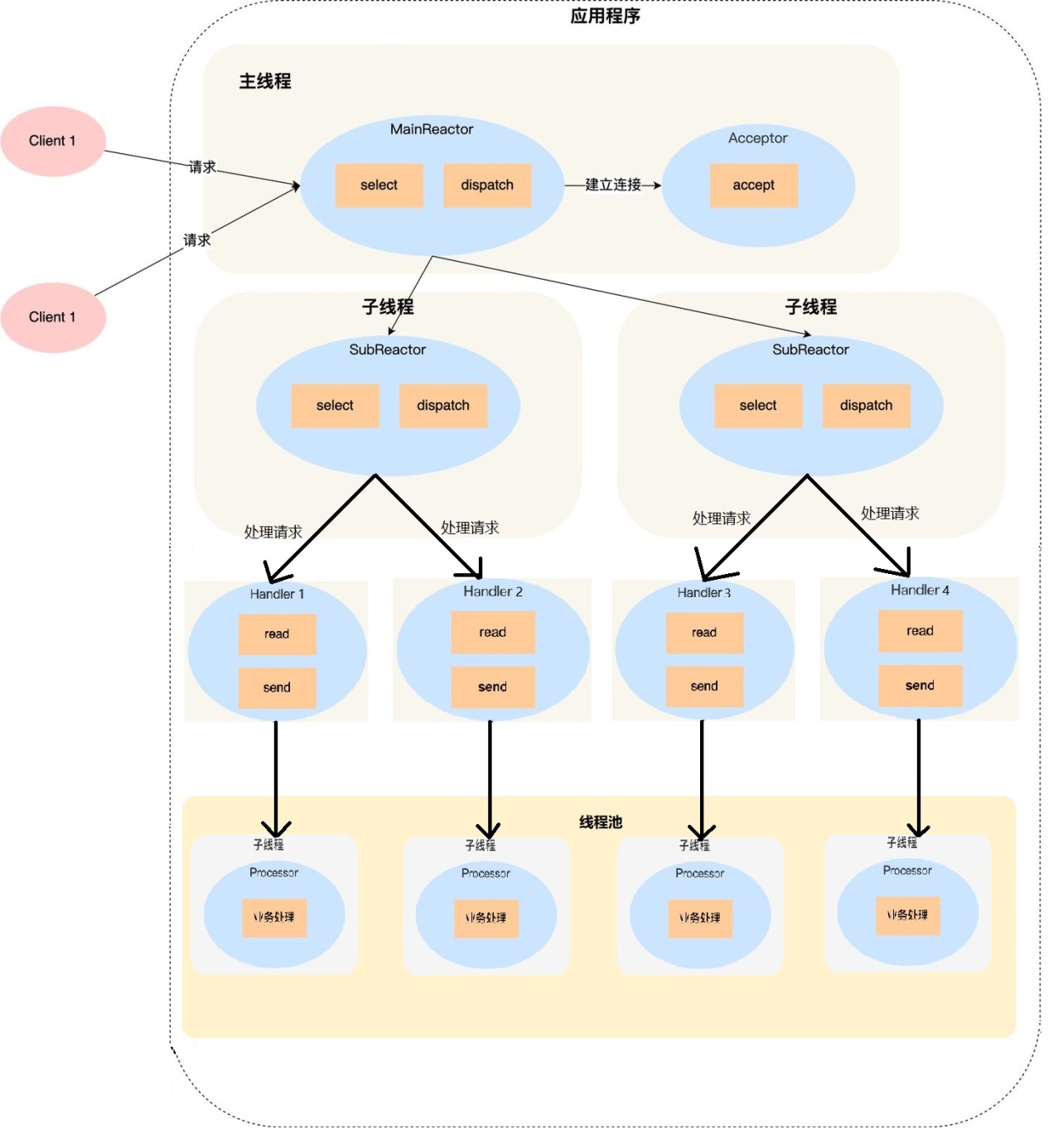
优点：

1. 多线程处理业务可以充分利用多核CPU的性能

缺点：

1. 多线程带来了资源竞争问题，加大了编程的复杂度
2. 单个Reactor进程/线程需要承载所有IO事件的监听和相应，成为了制约程序进一步提高性能的瓶颈
   * + 1. 多 Reactor，多进程/线程

为了进一步提高Reactor模式的性能，多Reactor多进程/线程方案出现了



上图中我们可以看到，Reactore线程/进程有多个，分为一主多从，即一个MainReactor和多个SubReactor

主要流程如下：

* + 1. 主线程中的 MainReactor 对象通过 select 监控连接建立事件，收到事件后通过 Acceptor 对象中的 accept 获取连接，处理连接事件，请注意，MainReactor仅负责监听处理新连接事件，不再监听IO读写事件；
    2. MainReactor将新的连接分配给某个子线程/线程中的 SubReactor 对象处理，SubReactor 对象将 MainReactor 对象分配的新连接加入 select 进行监听，并创建一个 Handler 用于处理连接的响应事件。
    3. 如果有新的事件发生时，SubReactor 对象会调用当前连接对应的 Handler 对象来进行响应。
    4. Handler 对象通过read读取数据后，将任务分发给线程池进行业务处理。
    5. 线程池中的 Processor 对象就进行业务处理，处理完后，将结果发给主线程中的 Handler 对象，接着由 Handler 通过 send 方法将响应结果发送给 client；

优点：

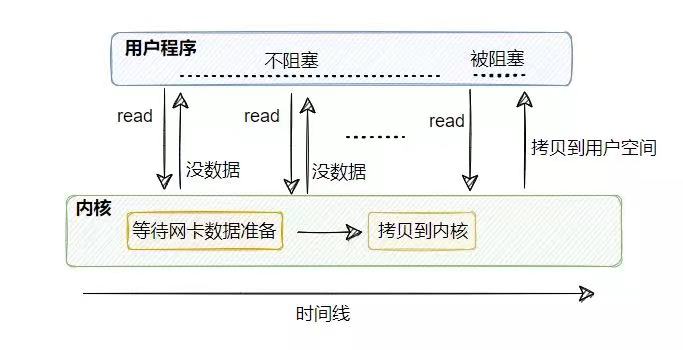
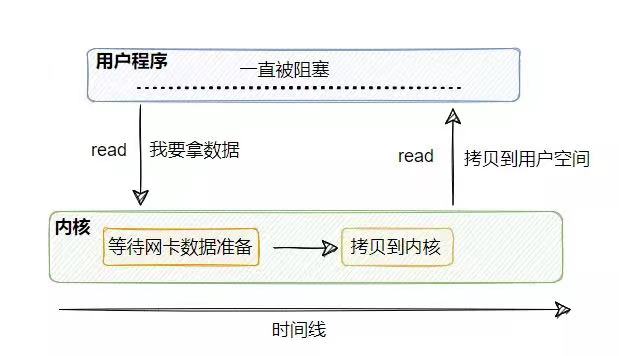
* + 1. 主线程和子线程分工明确，主线程只负责接收新连接，子线程负责完成后续的业务处理。
    2. 主线程和子线程的交互很简单，主线程只需要把新连接传给子线程，子线程无须返回数据，直接就可以在子线程将处理结果发送给客户端。

大名鼎鼎的两个开源软件 Netty 和 Memcache 都采用了「多 Reactor 多线程」的方案。采用了「多 Reactor 多进程」方案的开源软件是 Nginx，不过方案与标准的多 Reactor 多进程有些差异。

* + 1. Proactor模式

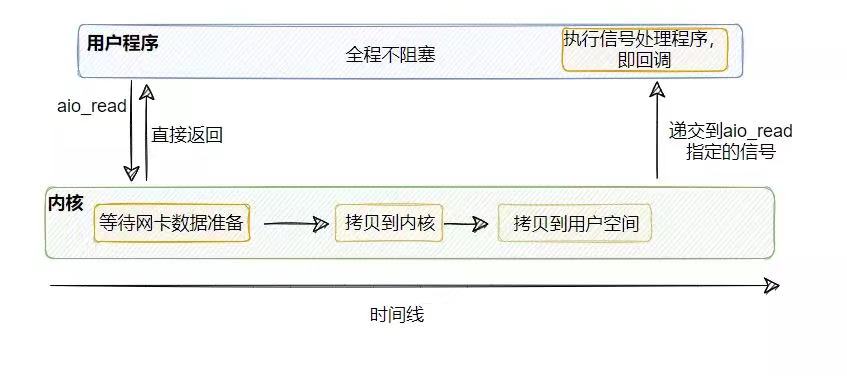
Reactor 是非阻塞同步网络模式，而 Proactor 是异步网络模式。

前面我们介绍过，IO分为：等待就绪和操作两个阶段，阻塞IO就是指我们的BIO，非阻塞即NIO，两者的模式如下图：

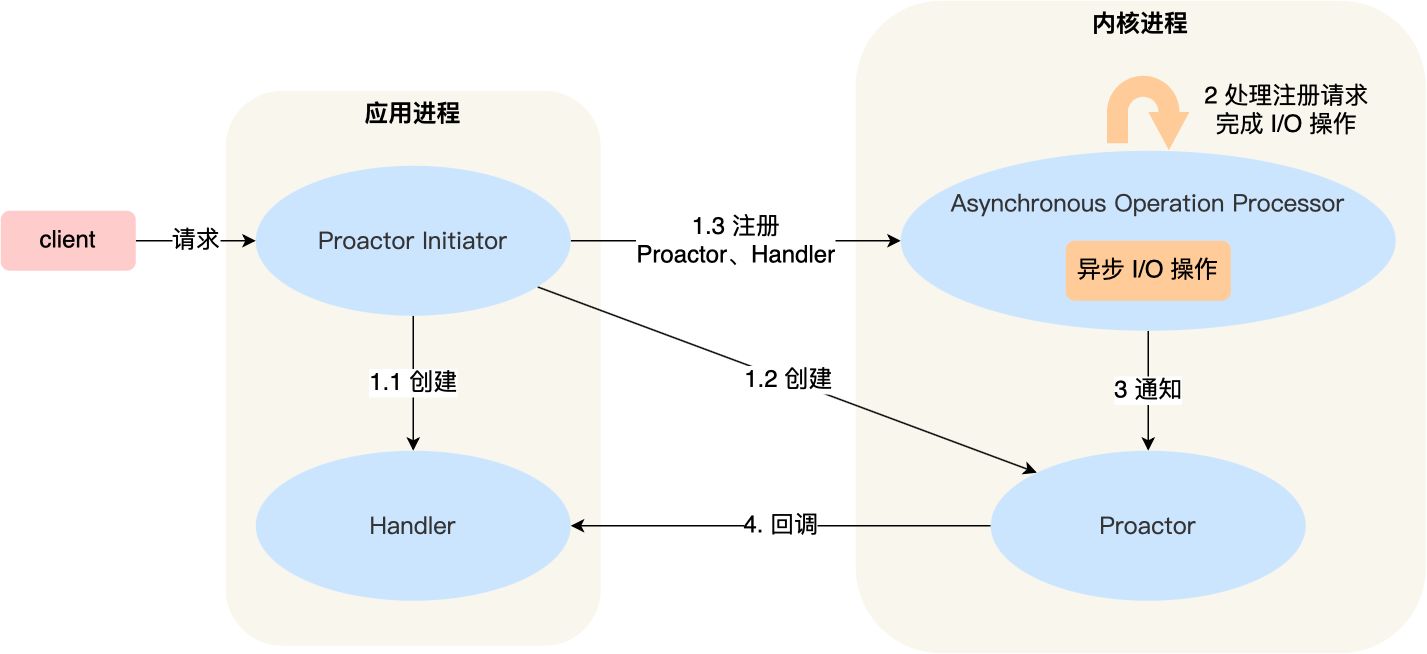


虽然我们的NIO是非阻塞的，但这只能说明等待就绪阶段不需要阻塞，而操作阶段，read在将数据从内核态拷贝到用户态时，仍然需要阻塞，这一点即使是IO多路复用也不能避免，因此这几种IO模型仍是同步模型，如果希望真正做到异步，还需要使用AIO，只有AIO可以做到在等待就绪和操作两个阶段都不阻塞

当我们发起 aio\_read （异步 I/O） 之后，就立即返回，内核自动将数据从内核空间拷贝到用户空间，这个拷贝过程同样是异步的，内核自动完成的，和前面的同步操作不一样，应用程序并不需要主动发起拷贝动作



Proactor 正是采用了异步 I/O 技术，所以被称为异步网络模型



1. Proactor Initiator 负责创建 Proactor 和 Handler 对象，并将 Proactor 和 Handler 都通过 Asynchronous Operation Processor 注册到内核；
2. Asynchronous Operation Processor 负责处理注册请求，并处理 I/O 操作；
3. Asynchronous Operation Processor 完成 I/O 操作后通知 Proactor；
4. Proactor 根据不同的事件类型回调不同的 Handler 进行业务处理；
5. Handler 完成业务处理；

1. 集合类

集合从大方向上来说分为Collection集合和Map集合

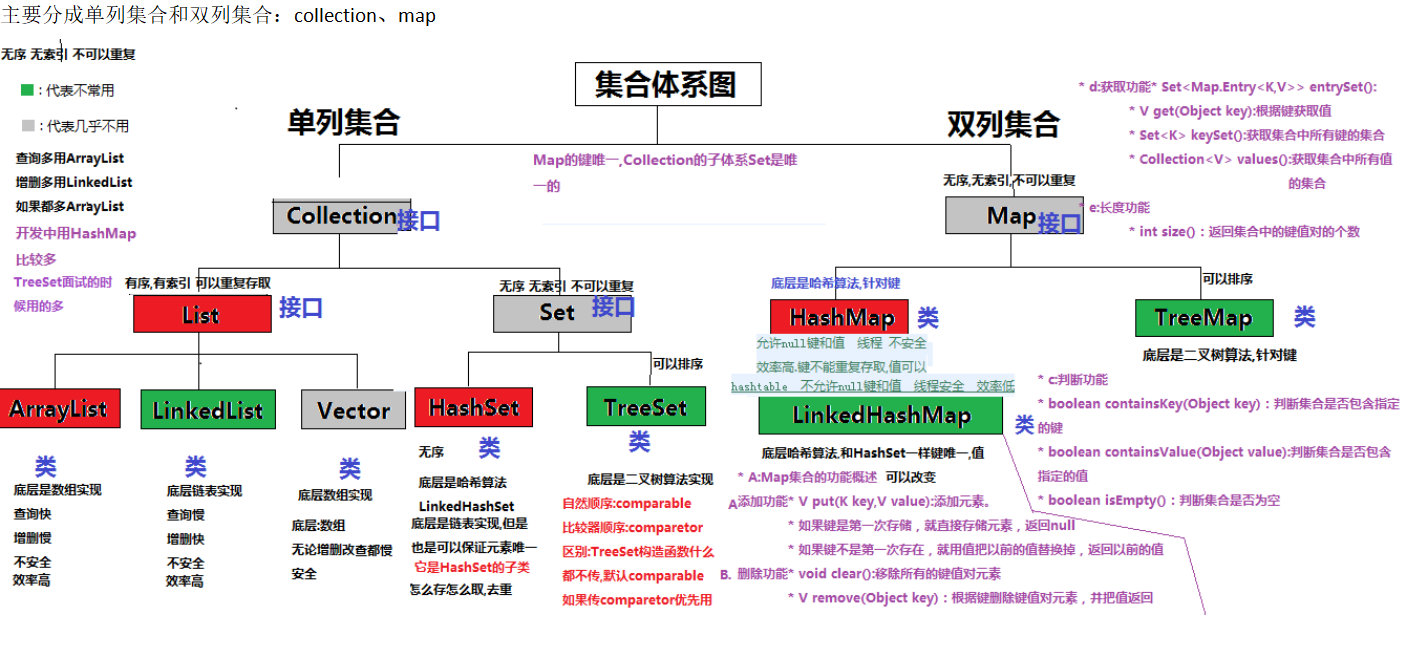
Collections又可以分为List和Set两类

List集合下有ArrayList、LinkedList、Vector、CopyOnWriteArrayList。

Set集合下有HashSet、LinkedHashSet、TreeSet、CopyOnWriteArraySet

Map集合下有HashMap、LinkedHashMap、TreeMap、HashTable、ConcurrentHashMap。

如下图



* 1. List
     1. ArrayList

arrayList的底层实现是数组，因此查询较快，时间复杂度为O(1)，增删由于需要移动元素，因此增删较慢，在第i个位置增删元素的时间复杂度为O(n-i)，属于线程不安全型集合，但效率较高

下面我们来分析一下arrayList的扩容机制

public boolean add(E e) {

//确认内部容量是否需要扩容，size是arraylist包含的元素个数，size+1是最小扩容量  
 ensureCapacityInternal(size + 1); // Increments modCount!!  
 elementData[size++] = e;  
 return true;  
}

private void ensureCapacityInternal(int minCapacity) {  
 ensureExplicitCapacity(*calculateCapacity*(elementData, minCapacity));  
}

private static int calculateCapacity(Object[] elementData, int minCapacity) {

//判断元素数组是否为空数组  
 if (elementData == *DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA*) {  
 //如果是空数组，返回默认容量和当前arraylist元素个数中最大的一个

return Math.*max*(*DEFAULT\_CAPACITY*, minCapacity);  
 }  
 return minCapacity;  
}

private void ensureExplicitCapacity(int minCapacity) {

// modCount表示list结构上被修改的次数。结构上的修改指的是那些改变了list的长度大小或者使得遍历过程中产生不正确的结果的其它方式。它被迭代器使用，通过它可以判断是否需要抛出ConcurrentModificationException异常  
 modCount++;

// 如果最小需要空间比当前空间大，则需要扩容  
 if (minCapacity - elementData.length > 0)  
 grow(minCapacity);  
}

private void grow(int minCapacity) {  
 // overflow-conscious code  
 int oldCapacity = elementData.length;

// 扩容至原来的1.5倍

int newCapacity = oldCapacity + (oldCapacity >> 1);

// 如果还不够，直接使用最小需要容量来作为扩容后的数组大小.  
 if (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;

// 如果新数组大小超过arraylist能存放的最大值，判断是否有内存溢出  
 if (newCapacity - *MAX\_ARRAY\_SIZE* > 0)  
 newCapacity = *hugeCapacity*(minCapacity);  
 // minCapacity is usually close to size, so this is a win:

// 将旧数组拷贝到新数组  
 elementData = Arrays.*copyOf*(elementData, newCapacity);  
}

arrayList的扩容机制：

当我们执行无参的new ArrayList()时，elementData实际上被赋值成一个空数组常量，当add第一个元素进去的时候，如果发现elementData是空数组常量则会将arrayList的大小初始化为10 *DEFAULT\_CAPACITY*，此后每次add元素都会先检查数组大小够不够，如果不够就需要扩容，每次扩容默认为原来的1.5倍，如果1.5倍不够，就会以最小的当前所需容量作为数组大小进行扩容，扩容后将旧数组中的内容拷贝到新数组

上述代码是1.8中的ArrayList，那么1.7中的arrayList和1.8有什么区别呢？

在JDK1.7中，使用ArrayList list = new ArrayList()创建List集合时，底层直接创建了长度是10的Object[]数组elementData，后续add的时候如果空间不足了再扩容，因此在开发中，建议使用带参构造器创建List集合:ArrayList list = new ArrayList(int capacity)，预估集合的大小，直接一次到位，避免中间的扩容，提高效率

改成1.8之后，new ArrayList()默认创建的是空数组，只有当add的时候才会扩容为10，这样做**延迟了数组的创建，节省了内存空间**。

* + 1. LinkedList

LinkedList查找慢，增删快，底层由链表实现，线程不安全性

在实际使用中：

在尾部增删(顺序增删)时，arraylist不一定慢，因为绝大多数的尾部增删，arraylist都不需要移动元素了，只有在扩容的时候才需要进行数组copy

在随机增删时，linkedList插入还需要查找到插入节点的前节点，这个查找部分是相当耗时的，经实际测试，大容量的随机增删，linkedList性能远低于ArrayList

因此在实际操作中，无论是查找还是增删，都是arrayList的综合性能更好

* + 1. Vector

底层是由数组实现的，增删改查都慢，唯一的优点是线程安全。

Vector的实现很简单，就是把所有的方法统统加上synchronized就完事了。你也可以不使用Vector，用Collections.synchronizedList把一个普通ArrayList包装成一个线程安全版本的数组容器也可以，原理同Vector是一样的，就是给所有的方法套上一层synchronized

* + 1. CopyOnWriteArrayList

众所周知，ArrayList线程不安全，那么如果想要线程安全，就必须使用Vector吗？CopyOnWriteArrayList可以帮助我们实现线程安全型list集合

vector最大的问题在于给读操作也加了锁，这是为了保证读写互斥，但是也产生了一个副作用—读读互斥，这降低了list的并发度，而CopyOnWriteArrayList仅对写操作加锁，对读操作不加锁，因此读读不互斥，而读写之间的并发问题，也采用了数组副本的方式来解决，这极大地提高了**多读少写场景**下的list并发度，同时又保证了线程安全

CopyOnWriteArrayList 类的所有可变操作（add，set 等等）都是通过创建底层数组的新副本来实现的，并且对于所有可变操作都是直接加锁，但是对于读取操作不加锁。当 List 需要被修改的时候，并不修改原有内容，而是对原有数据进行一次复制，将修改的内容写入副本。写完之后，再将修改完的副本替换原来的数据，而替换这一操作是原子性的，这样就可以保证写操作不会影响读操作了。

让我们来看一看它的源码：

首先是构造函数

final transient ReentrantLock lock = new ReentrantLock();  
// 存储数据的array数组，注意此处是用volatile修饰的，使用volatile可以使得**array引用**的线程中的副本在被修改后失效，保证array一旦被修改，对于其他线程的读操作是立即可见的

private transient volatile Object[] array;

final void setArray(Object[] a) {  
 array = a;  
}

public CopyOnWriteArrayList() {  
 setArray(new Object[0]);  
}

public CopyOnWriteArrayList(Collection<? extends E> c) {  
 Object[] elements;  
 if (c.getClass() == CopyOnWriteArrayList.class)  
 elements = ((CopyOnWriteArrayList<?>)c).getArray();  
 else {  
 elements = c.toArray();  
 // c.toArray might (incorrectly) not return Object[] (see 6260652)  
 if (elements.getClass() != Object[].class)  
 elements = Arrays.*copyOf*(elements, elements.length, Object[].class);  
 }  
 setArray(elements);  
}

下面我们看看add方法

public boolean add(E e) {  
 final ReentrantLock lock = this.lock;

//加锁  
 lock.lock();  
 try {

//获取原数组  
 Object[] elements = getArray();  
 int len = elements.length;

//拷贝出一个长度+1的新数组  
 Object[] newElements = Arrays.*copyOf*(elements, len + 1);

//数组的最后一位设置为入列元素  
 newElements[len] = e;

//原子操作，将老数组替换成新数组  
 setArray(newElements);  
 return true;  
 } finally {  
 lock.unlock();  
 }  
}

public void add(int index, E element) {  
 final ReentrantLock lock = this.lock;

//加锁  
 lock.lock();  
 try {

//获取原数组  
 Object[] elements = getArray();  
 int len = elements.length;

//判断数组是否越界  
 if (index > len || index < 0)  
 throw new IndexOutOfBoundsException("Index: "+index+  
 ", Size: "+len);  
 Object[] newElements;

//计算要移动的位数  
 int numMoved = len - index;  
 if (numMoved == 0)

//如果index是在数组尾部添加  
 newElements = Arrays.*copyOf*(elements, len + 1);  
 else {

//new一个新数组  
 newElements = new Object[len + 1];

//拷贝旧数组的前半部分到新数组  
 System.*arraycopy*(elements, 0, newElements, 0, index);

//拷贝旧数组的后半部分到新数组  
 System.*arraycopy*(elements, index, newElements, index + 1,  
 numMoved);  
 }

//将新数组空出来的index位赋值为新入列元素  
 newElements[index] = element;

//原子操作，将老数组替换成新数组  
 setArray(newElements);  
 } finally {  
 lock.unlock();  
 }  
}

我们可以看到，add方法通过ReentrantLock保证同一时刻最多只有一个线程向list中添加元素，并且不是直接往数组中添加元素，而是开辟新数组，把元素插入新数组，再用新数组替换旧数组。

既然已经使用ReentrantLock保证了同一时刻只能有一个线程执行这段代码，那为什么还要开辟新数组呢？因为不开辟新数组的话，还会有读写的并发问题，volatile只能保证array引用地址被修改后是可见的，不能保证数组元素被修改后对其他线程仍可见

因此，只有在赋值成功时新元素才算真正add进去，而赋值操作**setArray(newElements)**是原子性的(参考[原子性](#赋值操作是原子性的))。

在setArray执行之前所有的读操作都是在读取旧数组的数据，而setArray这一原子性操作执行后，array的缓存失效，强制其他线程去主存中读取新数组，这样就解决了读写的并发问题

也就是说，并发的三个特性：原子性(赋值本身就是原子性的)、可见性和有序性(通过volatile保证)都得到了保障，因此CopyOnWriteArrayList是线程安全型集合

CopyOnWriteArrayList虽然底部也是数组实现，但是没有扩容这个说法。因为每次add都会开辟新的数组。况且每次add都会加锁，所以效率是比较低的。因此CopyOnWriteArrayList的add效率比vector还低，但在多线程中get方法的效率远高于vector

下面我们看一下get方法

private E get(Object[] a, int index) {  
 return (E) a[index];  
}public E get(int index) {  
 return get(getArray(), index);  
}

可以看出get方法是不加锁的

CopyOnWriteArrayList的优点主要有两个：

1. 线程安全
2. 大大的提高了“读”操作的并发度（相比于Vector）

缺点也很明显：

1. 每次“写”操作都会开辟新的数组，浪费空间
2. 无法保证实时性，因为“读”和“写”不在同一个数组，且“读”操作没有加互斥锁，所以不能保证**强一致性**，只能保证**最终一致性**
3. add/remove操作效率低，既要加锁，还要拷贝数组

所以CopyOnWriteArrayList比较适合**读多写少**的场景。

* 1. Map

JDK1.0时的集合类都是线程安全型的，如：

HashSet：线程不安全，效率高，基于HashMap实现

TreeMap：线程不安全，效率高，基于红黑树的Map实现

TreeSet ：线程不安全，效率高，基于TreeMap实现

JDK1.5前的线程安全型集合，全部都是同步集合(如Vector和HashTable)，主要通过synchronzied实现同步，使得每一次都只能有一个线程访问容器，这严重影响了性能。到了java5之后，为了改进同步容器的性能，JDK提供了并发集合

常见的并发集合有：ConcurrentHashMap、ConcurrentLinkedQueue、CopyOnWriteArrayList、CopyOnWriteArraySet，下面我们将深入分析这几种并发集合

* + 1. HashMap
       1. JDK1.7中的HashMap
          1. java7中HashMap的原理

我们先来看看hashmap的结构：

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable{ *//bucket数组的初始化大小，默认16*static final int *DEFAULT\_INITIAL\_CAPACITY* = 1 << 4; // aka 16 *//bucket数组的最大值  
//这里可以看出，hashcode范围是int的取值范围，而bucket数组的最大值明显小于hashcode的范围，因此不能直接使用hashcode作为数组下标*static final int *MAXIMUM\_CAPACITY* = 1 << 30;  
*//负载因子默认值*static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;  
*//bucket数组*static final Entry<?,?>[] *EMPTY\_TABLE* = {};transient Entry<K,V>[] table = (Entry<K,V>[]) *EMPTY\_TABLE*;  
*//键值对的数量*transient int size;  
*//扩容的阈值，等于 capacity \* loadFactor，其中capacity是bucket数组实际长度*

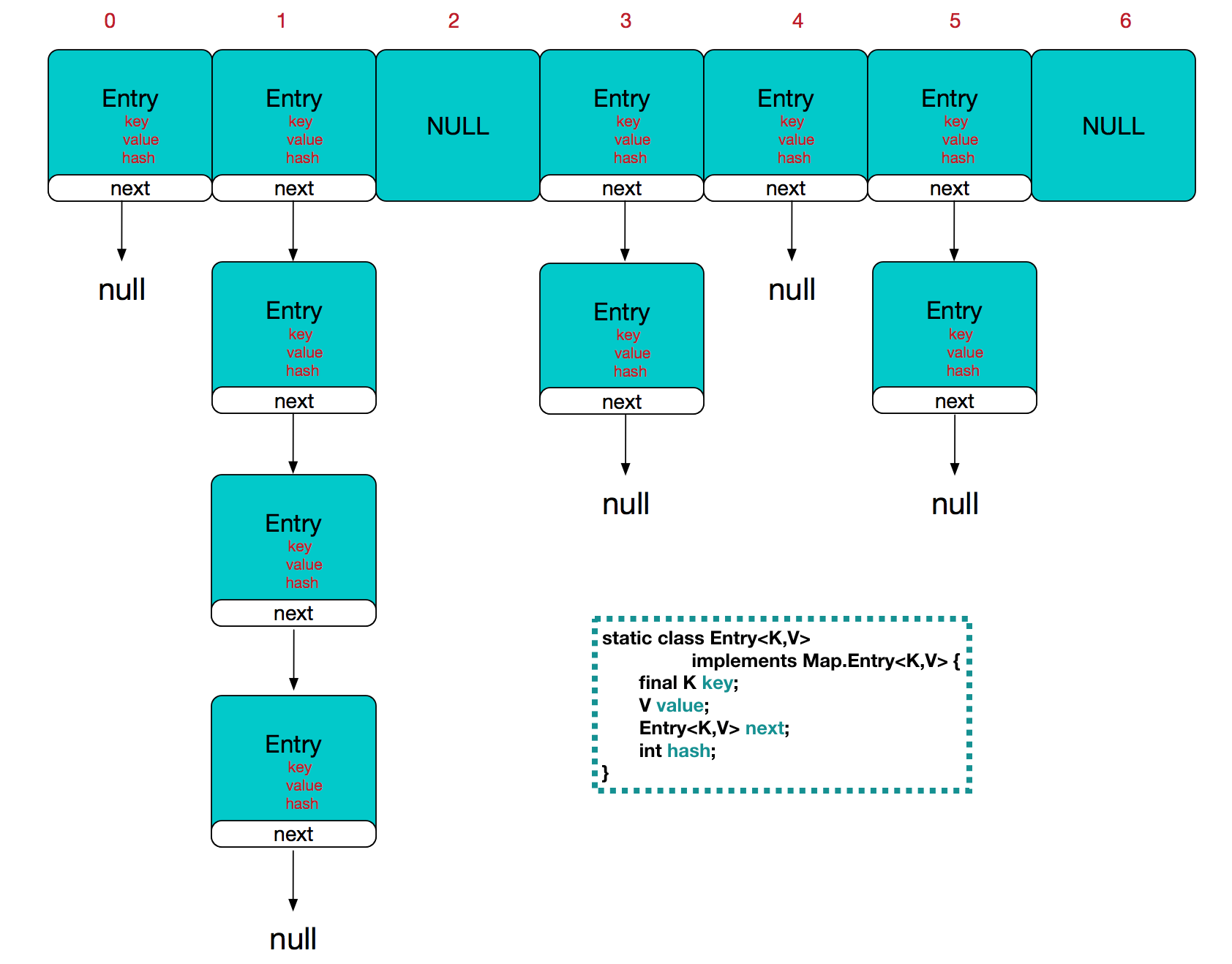
*//如果HashMap.size >= Capacity \* LoadFactor成立，HashMap将进行扩容。其中，HashMap.Size是HashMap在Put操作之前实际容纳的键值对Entry数量*  
int threshold;  
*//负载因子*final float loadFactor;  
*//修改计数器，用于*[***快速失败机制***](#快速失败机制与安全失败机制)transient int modCount;

static class Entry<K,V> implements Map.Entry<K,V> {  
 final K key;  
 V value;

*//指向下一节点的引用，用于实现链表*  
 Entry<K,V> next;

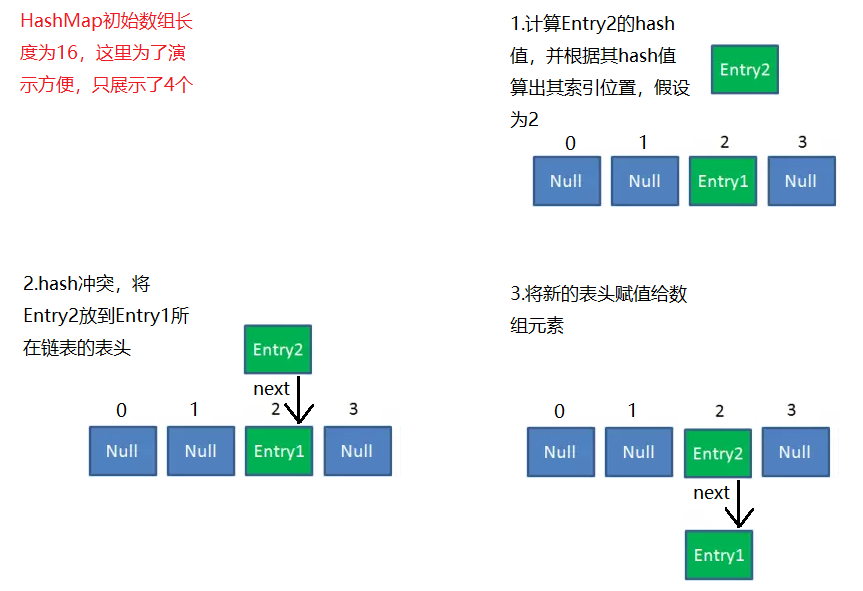
*//存放当前key的hash值，这个hash值是被hash函数扰动过的hash值，参见*[***hash扰动函数***](#hash扰动)  
 int hash;Entry(int h, K k, V v, Entry<K,V> n) {  
 value = v;  
 next = n;  
 key = k;  
 hash = h;  
 }

}



如上图所示HashMap由一个Entry数组(bucket数组)和多个Entry链表组成，其主干是一个数组(bucket数组)，数组(bucket数组)的每个元素代表一个hash值，其存储的是每个链表的表头元素，所有发生hash冲突的Entry都将被放到链表中。一个好的哈希函数应该要尽量平均元素在数组中的分布，减少哈希冲突从而减小链表的长度。链表的长度越长，意味着在查找时需要遍历的结点越多，哈希表的性能也就越差。

当我们试图使用put(key,value)存储对象到HashMap中时，首先执行hash(key) & (length-1)【与n取模实际上就是与n-1进行按位与操作，因此上述操作= hash(key)%length，但是计算效率高10倍】，hash(key)是算出key的hash值，然后对hash值和bucket数组长度-1进行&操作，最后得出该Entry节点在bucket数组中的索引位置，如果当前位置已经有Entry(即不为null)，那么我们就说HashMap发生了hash冲突，JDK1.7的HashMap是采用链表的方式解决Hash冲突的，他会将新的Entry放到当前bucket位置上链表的头部，然后再将新Entry赋值到当前bucket位置上，如图所示：



当我们试图使用get()方法获取Entry时，依然是通过hash(key) & (length-1)找到其bucket位置，然后循环链表，通过key.equals()方法找到我们要找的键值对。如上图我们如果希望查找Entry1的键值对，首先我们对key1算出hash，然后计算出其bucket位置为2，然后遍历链表，通过equals找到对应的Entry

* + - * 1. HashMap的扩容

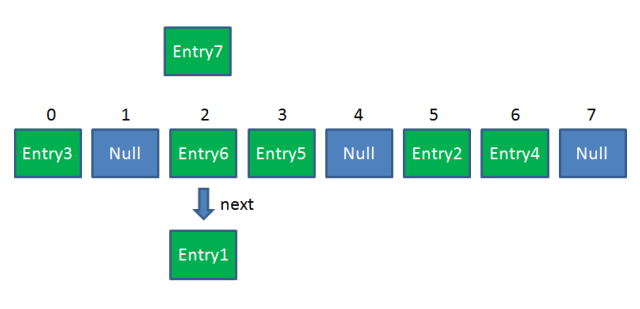
在HashMap中，有三个重要的属性：

**capacity：当前数组容量，始终保持 2^n，默认初始大小为16，可以扩容，扩容后数组大小为当前的 2 倍。**

**loadFactor：负载因子，默认为 0.75。**

**threshold：扩容的阈值，等于 capacity \* loadFactor**

HashMap的容量是有限的。当经过多次元素插入，使得HashMap达到一定饱和度时，Key映射位置发生冲突的几率会逐渐提高，其性能也急剧下降。这时候，HashMap需要扩展它的长度，也就是进行**Resize（扩容是非常消耗性能的，提前预估 HashMap 的大小最好，尽量的减少扩容带来的性能损耗）**。



在HashMap执行put方法前会有一个判断，如果**HashMap.size  >=** **Capacity \* LoadFactor**成立，HashMap将进行扩容，其中，**HashMap.Size**是HashMap在Put操作之前实际容纳的键值对Entry数量，上图中是6，**Capacity**是HashMap数组的长度，数组长度应当是2的幂，上图中**Capacity**为8，**LoadFactor**是负载因子，默认值为0.75f，当上图的HashMap再次执行put操作时并将要执行结束前，6>=8\*0.75f成立，因此上图中的HashMap应当扩容

那么扩容做了什么呢？

**1. 创建一个新的Entry空数组，长度是原数组的2倍。**

**2. 遍历原Entry数组，把所有的Entry重新Hash到新数组。为什么要ReHash呢？因为长度扩大以后，Hash寻址的规则也随之改变。**

void resize(int newCapacity) {  
 Entry[] oldTable = table;  
 int oldCapacity = oldTable.length;  
 if (oldCapacity == *MAXIMUM\_CAPACITY*) {  
 threshold = Integer.*MAX\_VALUE*;  
 return;  
 }  
  
 Entry[] newTable = new Entry[newCapacity];

*//rehash，将旧数组中的Entry重新转换到扩容后的新数组中.*  
 transfer(newTable, initHashSeedAsNeeded(newCapacity));  
 table = newTable;  
 threshold = (int)Math.*min*(newCapacity \* loadFactor, *MAXIMUM\_CAPACITY* + 1);  
}

*//将旧数组中的Entry重新转换到扩容后的新数组中.*void transfer(Entry[] newTable, boolean rehash) {  
 int newCapacity = newTable.length;  
 for (Entry<K,V> e : table) {

//对原table数组中的每一条链表进行循环reHash  
 while(null != e) {

//next引用指向原table数组中的e的下一个结点Entry  
 Entry<K,V> next = e.next;

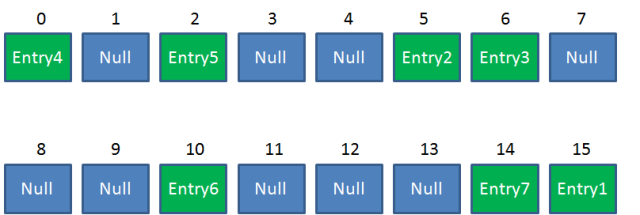
//计算e引用指向的Entry在新table数组中的位置   
 if (rehash) {  
 e.hash = null == e.key ? 0 : hash(e.key);  
 }  
 int i = indexFor(e.hash, newCapacity);

//如果新table的i位置上已有Entry了，就要将e的next域指向该位置上的Entry，实际上是让e成为链表的表头节点   
 e.next = newTable[i];

//将e，也就是链表的头节点，放到新table数组的i位置  
 newTable[i] = e;  
 //将e指向原table数组的链表中的下一个节点，进行下一次循环

e = next;  
 }  
 }  
}

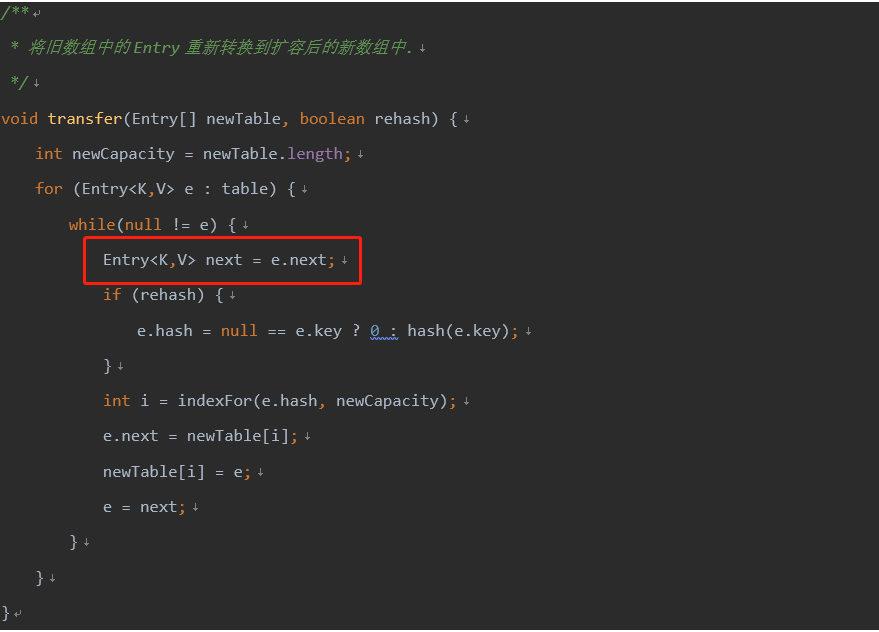
上图的HashMap在经历一次扩容后，会变成下面的样子：



* + - * 1. 并发下的阴云

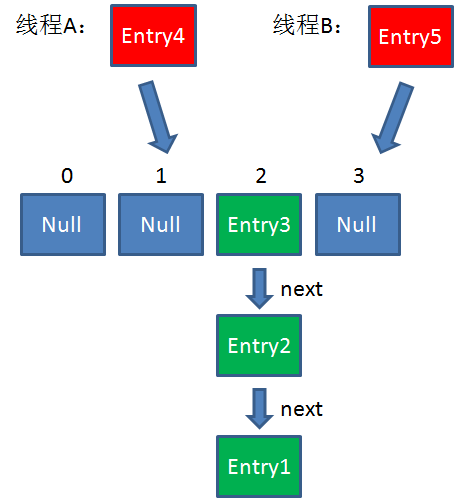
一切看起来似乎都那么完美，然而，此时晴朗的天空飘来了一朵乌云——并发

我们首先来看一下rehash的函数：

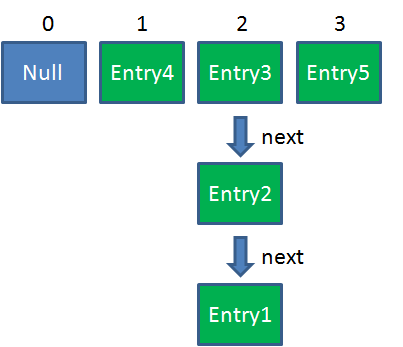


我们可以看到，java7的rehash是采用**头插法**进行的，即发生hash冲突的新节点将被放到链表的头节点上

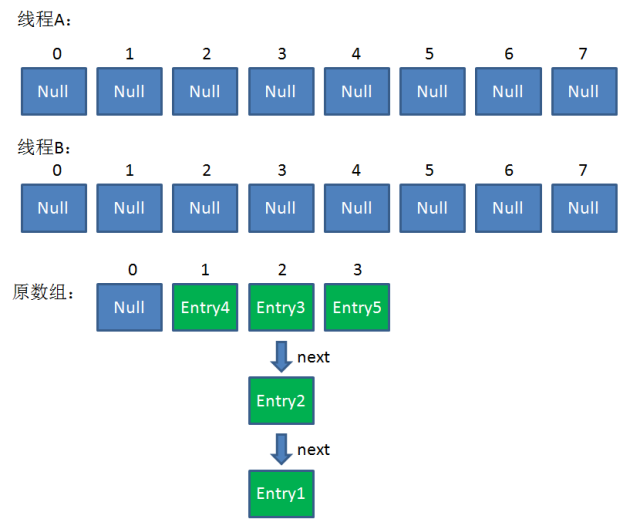
假设现有一个HashMap正处于扩容边缘，此时来了两个线程同时对其进行put操作



上面我们提到过，put操作首先会添加，然后添加完成，再判断是否要扩容，那么就会变成下图这样：

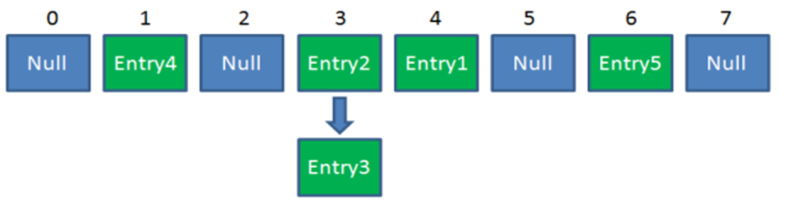
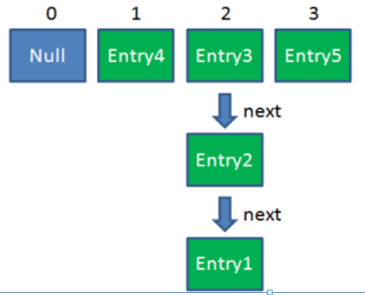


此时，线程A和线程B进行判断，发现需要扩容，两条线程同时准备对HashMap进行扩容，各自创建一个长度为原来两倍的数组



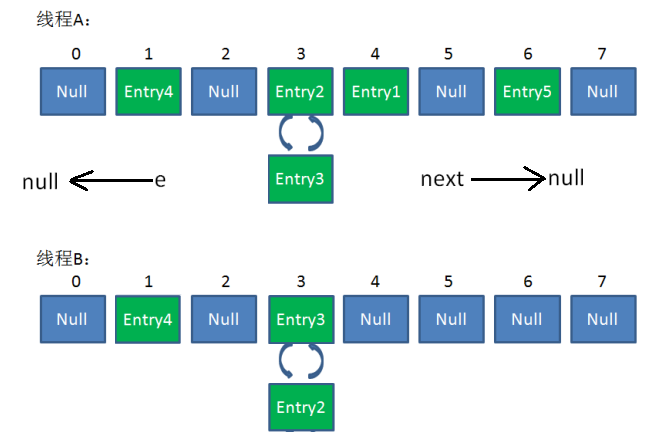
接着，他们各自准备遍历原数组，进行ReHash操作

假设rehash之后，Entry2仍和Entry3发生hash冲突，那么一个正常的rehash就会将map变成下面的样子，我们可以看到，头插法导致了Entry3和Entry2的顺序颠倒，如下图



假设线程B刚执行完上图transfer()函数中红框里的这行代码，即刚刚把e引用指定为Entry3节点，next引用指定为Entry2就被挂起；

然后A线程从开始扩容一直执行到结束rehash，对于A线程来说，**它将链表颠倒了**，即Entry2.next指向了Entry3。但是此时B线程获得cpu继续执行，B会继续执行头插法操作，即e.next=table[i]，即e.next=表头，即Entry3.next=Entry2。此时B线程使得Entry3.next=Entry2，A线程导致Entry2.next=Entry3，这就形成了循环链表



最终，线程B仍会完成HashMap的扩容，只不过，在Entry2和Entry3之间，形成了一个环。并发下的Put操作在HashMap扩容的过程中埋下了一颗雷，当尝试调用Get查找一个不存在的Key，而这个Key的Hash结果恰好等于3的时候，它会不断地寻找next节点上的Entry是否符合key，然而始终占不到，并且由于位置3带有环形链表，所以程序将会进入死循环！

* + - * 1. 为什么每次扩容数组长度都必须是2的n次幂？

1. HashMap新添加的元素是通过 **(n - 1) & hash**进行的，(n-1)&hash实际上也是hash&(n-1)，这里的n指的是数组长度，只有当数组长度为2的幂次方时，**(n - 1) & hash**才等价于hash%n(参见[与运算](#与运算与2的n次幂)一节)，即只有在n是2的幂次方的情况下，与运算能够等价于取余预算，而取余运算可以减少冲突次数；
2. 如果 length 为 2 的次幂 则 length-1 转化为二进制必定是 11111……的形式，在与 hash 的二进制与操作效率会非常的快，而且空间不浪费；如果 length 不是 2 的次幂，比如 length 为 15，则 length - 1 为 14，对应的二进制为 1110，再与hash与操作，最后一位都为 0 ，永远不会得出 0001，0011，0101，1001，1011，0111，1101 这类最后一位都为1的结果，那这几个最后一位为1的位置永远都不能存放元素了，空间浪费相当大，更糟的是这种情况中，数组可以使用的位置比数组长度小了很多，这意味着进一步增加了碰撞的几率，减慢了查询的效率！这样就会造成空间的浪费。

那么hashmap为什么不直接使用hashcode作为table数组的index，而是采用**(n - 1) & hash**，即hash%n作为table数组的下标呢？

1. 首先hashcode返回的是int整数类型，其范围为-(2 ^ 31)~(2 ^ 31 - 1)，他是存在负值的，而数组index不能有负值
2. 其次，hashcode的范围为-(2 ^ 31)~(2 ^ 31 - 1)，约有40亿个映射空间，而HashMap的容量范围是在16（初始化默认值）~2 ^ 30(即**1 << 30**)，HashMap通常情况下是取不到最大值的，并且设备上也难以提供这么多的存储空间，从而导致通过hashCode()计算出的哈希值可能出现数组越界；
   * + - 1. java7中的put方法

public V put(K key, V value) {

*//判断当前数组是否需要初始化*  
 if (table == *EMPTY\_TABLE*) {  
 inflateTable(threshold);  
 }

*//如果key是null，放一个空值进去，hashmap允许null值，所有的null key都会被存储到table[0]的链表中，一个hashmap只能储存一个null key*  
 if (key == null)  
 return putForNullKey(value);

*//对key的hash值进行扰动，计算出hashcode*  
 int hash = hash(key);

*//根据hash值和bucket数组长度计算出该节点在bucket数组上的index索引位置，其实就是hash & (length-1)，参见*[***为什么不直接使用hashcode作为table数组***](#为什么不直接使用hashcode作为bucket数组的下标)  
 int i = *indexFor*(hash, table.length);

*//遍历链表*  
 for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
 Object k;

*//查看是否有节点的key和新增的key相等，如果有则直接覆盖*  
 if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  
 V oldValue = e.value;  
 e.value = value;  
 e.recordAccess(this);  
 return oldValue;  
 }  
 }

*//修改计数+1，*[***快速失败机制***](#快速失败机制与安全失败机制)  
 modCount++;

*//创建节点*  
 addEntry(hash, key, value, i);  
 return null;  
}

*//创建节点方法*

void addEntry(int hash, K key, V value, int bucketIndex) {

*//判断是否需要扩容*  
 if ((size >= threshold) && (null != table[bucketIndex])) {

*//扩容为原来的两倍*  
 resize(2 \* table.length);

*//扩容后需要重新计算hash值和index位置*  
 hash = (null != key) ? hash(key) : 0;  
 bucketIndex = *indexFor*(hash, table.length);  
 }

*//创建节点*  
 createEntry(hash, key, value, bucketIndex);  
}

* + - * 1. java7中的get方法

public V get(Object key) {

*//如果key是null，从table[0]的链表中查找key为null的节点，找到了就直接返回，因为hashmap只能存一个null key*  
 if (key == null)  
 return getForNullKey();

*//根据key获取值*  
 Entry<K,V> entry = getEntry(key);  
 return null == entry ? null : entry.getValue();  
}

*//根据key获取值*

final Entry<K,V> getEntry(Object key) {  
 if (size == 0) {  
 return null;  
 }

*//计算key的hash值*  
 int hash = (key == null) ? 0 : hash(key);

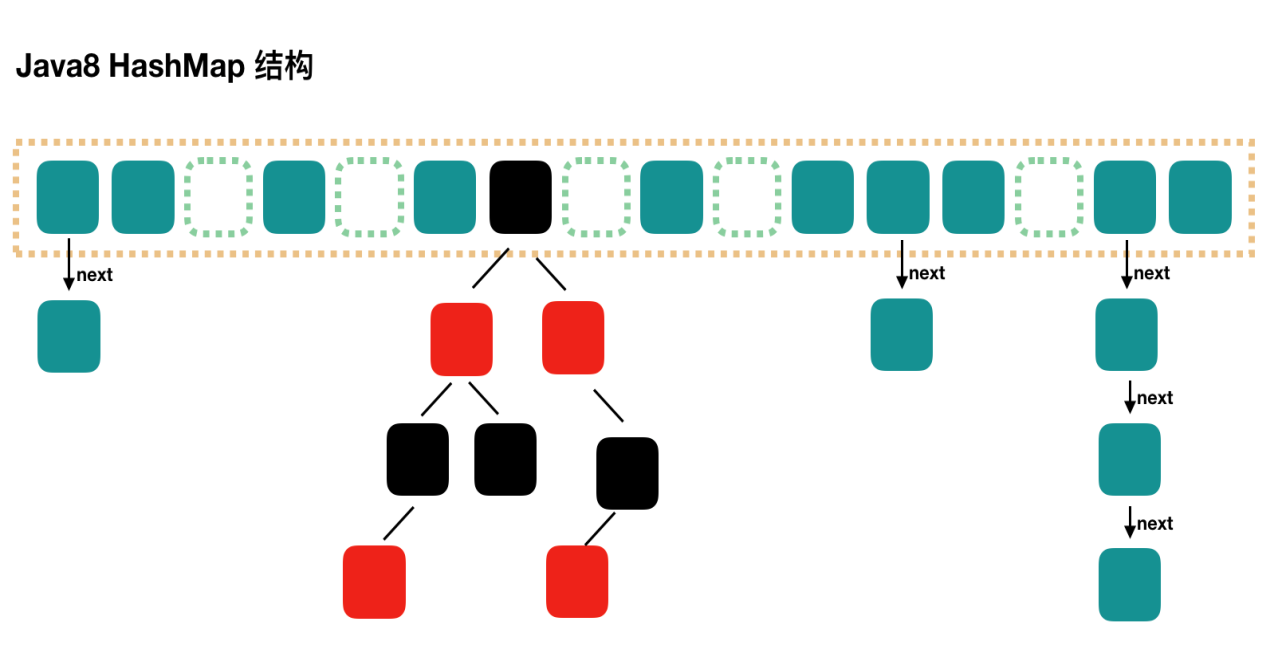
*//遍历key对应bucket上的那根链表*  
 for (Entry<K,V> e = table[*indexFor*(hash, table.length)] ; e != null ; e = e.next) {  
 Object k;

*//找到节点e后直接返回*  
 if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))  
 return e;  
 }  
 return null;  
}

* + - 1. JAVA 8中的HashMap

JAVA 8对HashMap进行了优化，最大的一点就是引入了”红黑树”。在JAVA 7中，HashMap在通过key来get一个value时，是先算出其key的hash值，然后通过hash值定位到数组的某一个元素上，该元素是链表的头节点，然后再遍历链表，一个一个查找和key相等的节点，查找的时间复杂度取决于链表的长度，为O(n)。在最坏的情况下，HashMap中所有的Entry节点都被映射到同一个数组bucket上，此时HashMap将退化成一个链表，因此过多的Hash碰撞将给HashMap的性能带来灾难性的影响

为了解决这种在糟糕场景下性能下降的问题，JAVA 8引入了红黑树，它的结构是这样的：



当然，上述结构不可能出现，因为节点这么多早就扩容了,这里只是做一个示例

当链表中的元素超过了 8 个以后，java8中的hashMap会将链表转换为红黑树。链表查找的时间复杂度为O(n)，而红黑树一直是O(logn)，这样就大幅提高了HashMap在大量发生hash碰撞情况下的查找效率。

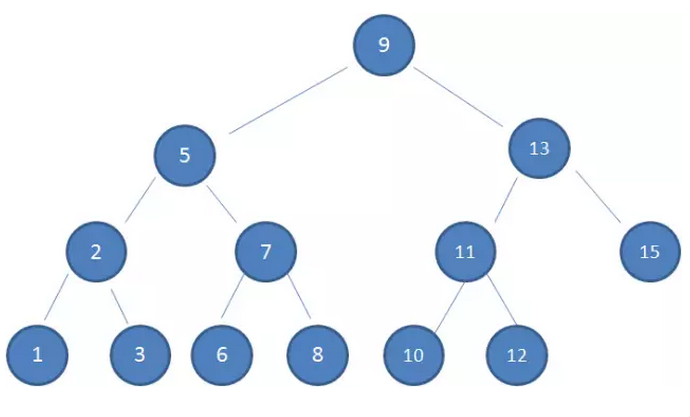
* + - * 1. 二叉查找(搜索)树

在理解红黑树之前，需要先了解二叉查找树，那么什么样的树是二叉查找树呢？

1.左子树上所有结点的值均小于或等于它的根结点的值。

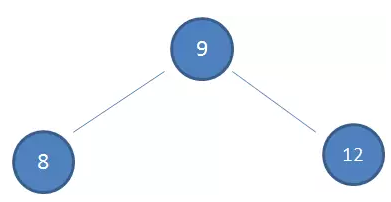
2.右子树上所有结点的值均大于或等于它的根结点的值。

3.左、右子树也分别为二叉查找树

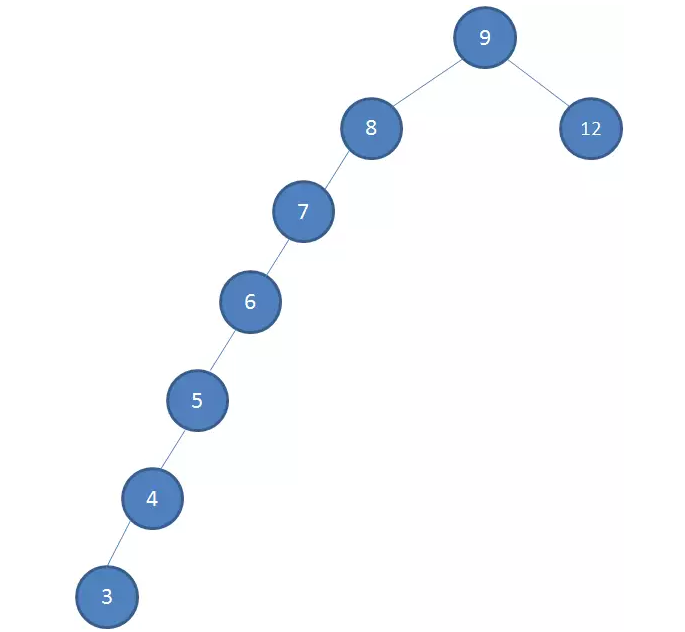


这样的结构可以帮助我们快速查找到自己想要的节点，比如我们希望查找值为12的节点，首先9<12，所以看右子树，13>12，所以看左子树，11<12所以看右子树，成功找到值为12的节点。二叉查找树，其最坏情况下的查找次数等于树的高度，这极大提高了节点的查询效率。当然了，在插入节点时，也是类似的道理。

但是，在一些极端情况下，二叉查找树可能退化到接近链表的程度，比如下面这种情况



上图是一个8，9，12的二叉搜索树，现在向这棵树中插入3,4,5,6,7，结果如下图：



在这种情况下，二叉查找树的查找效率将大打折扣，为了解决这种极端情况下性能损耗严重的问题，红黑树应运而生

* + - * 1. 红黑树

首先，红黑树是一棵二叉查找树，它在每一个节点上增加了一个存储域用于表示当前节点的颜色。它满足所有二叉查找树的条件，除此之外，他还要满足以下条件：

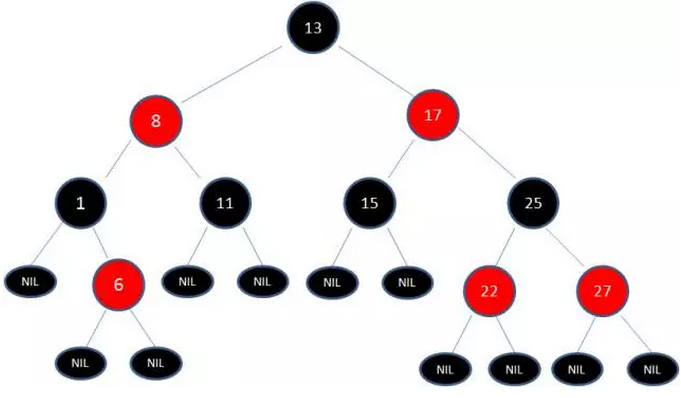
1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

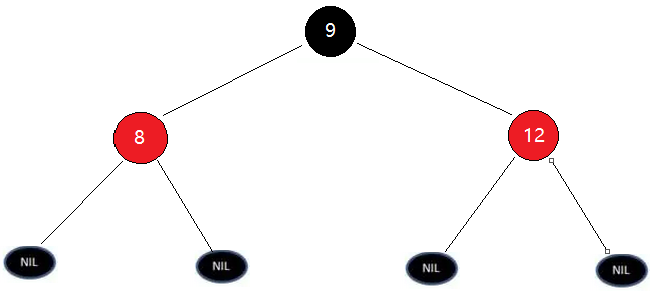
5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。



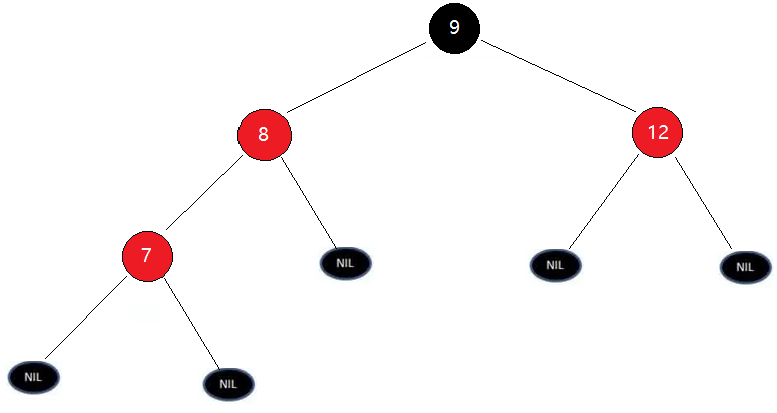
通过第四、五点的约束，我们可以知道，从根节点到叶子节点的所有路径中，最长路径是红黑交替的路径，最短路径是全黑的路径，因此**从根到叶子的所有路径中，没有哪条路径会比别的路径长出2倍。所以我们可以说红黑树是近似平衡的**

再回到我们上面的二叉搜索树的最差情况，

首先是8、9、12，然后依次向其中加入7，6，5，4，3

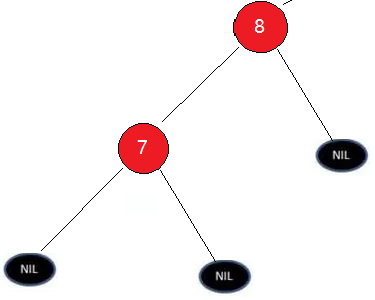
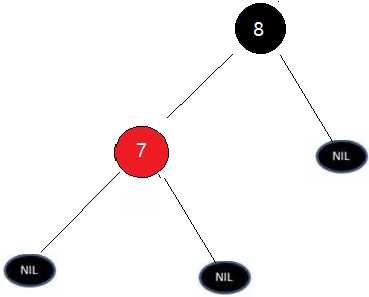


7应当在8的左边，为了保证，9经过8，7到叶子节点只有两个黑色节点(第5条约束)，因此新加入了7应当是红色的，但这又与第4条约束向矛盾(不能有两个相连的红色节点)，因此我们必须对红黑树进行调整。调整的方式有很多，主要分为变色和旋转，而旋转又可以分为左旋转和右旋转(ps:一般来说都是直接把插入的节点染成红色，这样只要满足4即可)

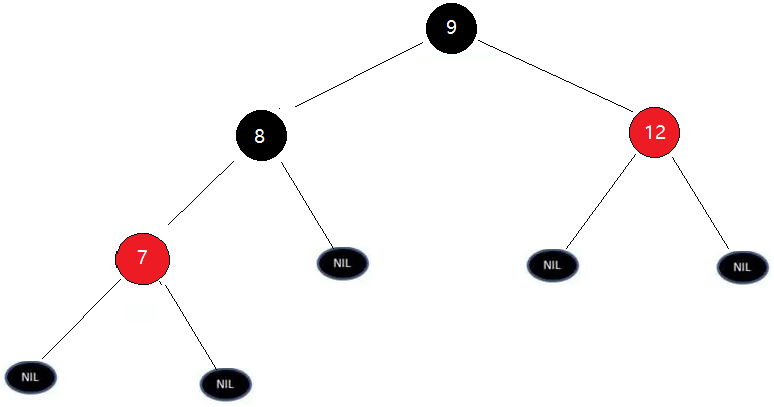


变色

变色是为了维护红黑树的结构，尝试将节点颜色进行转变的一种操作

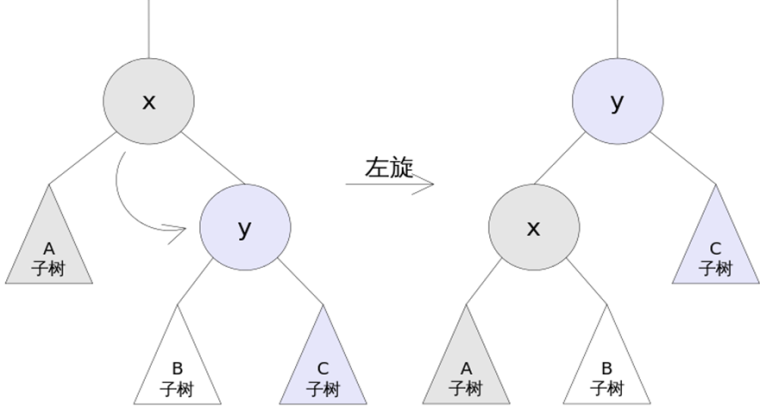
 转变为 

但是这样一来，又违背了第5条约束，从根节点到叶子节点出现了3个黑色节点，出现了连锁变化，因此还需要修改树节点的其他地方

转变为

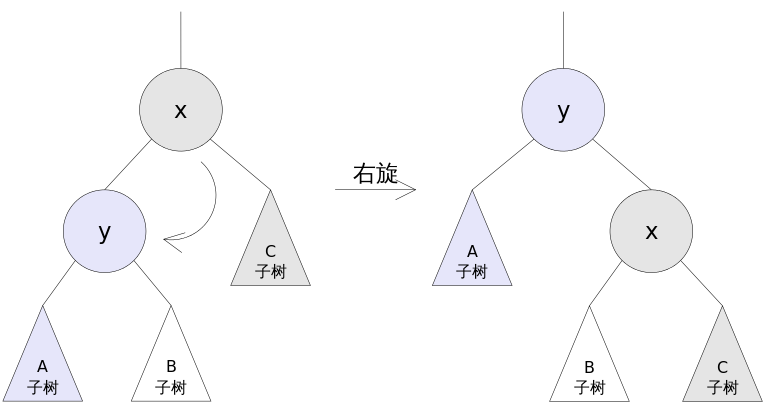
左旋转

左旋转：逆时针旋转红黑树的两个节点——父节点x和右子节点y，x被y取代，y成为父节点，x成为y的左子节点，同时，y原来的左子节点将变为x的右子节点



右旋转

右旋转：顺时针旋转红黑树的两个节点——父节点x和左子节点y，x被y取代，y成为父节点，x成为y的右子节点，同时，y原来的右子节点将变为x的左子节点



* + - * 1. 源码解析(内部结构和put过程)

内部结构

为了加入红黑树，java8提供了一个静态内部类TreeNode，我们可以看到它的构造函数提供了初始化Entry的方法

static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {

TreeNode<K,V> parent; // red-black tree links  
 TreeNode<K,V> left;  
 TreeNode<K,V> right;  
 TreeNode<K,V> prev; // needed to unlink next upon deletion  
 boolean red;  
 TreeNode(int hash, K key, V val, Node<K,V> next) {  
 super(hash, key, val, next);  
 }

}

put过程

接下来看一下put方法

resize过程

* + - 1. java8相比于java7的改动
         1. 引入红黑树

当链表中的元素大于 8 个以后且bucket数组长度大于64（如果长度不超过64则进行扩容），java8中的hashMap会将链表转换为红黑树，数量小于6时转换回链表

选择8的原因是因为红黑树的平均查找长度是log(n)，长度为8的时候，平均查找长度为3，如果继续使用链表，平均查找长度为8/2=4，这才有转换为树的必要。选择6转回链表而非8的原因是为了避免频繁转换

使用红黑树后，当链表过长时，查找性能从O(n)变为了O(logn)

* + - * 1. 修改构造函数中计算初始化数组容量方式

由于hashmap要求bucket数组容量必须为2^n，因此用户传入任意一个非2^n的数字时，都需要找到>=该数字的最小2^n。

在jdk1.7中

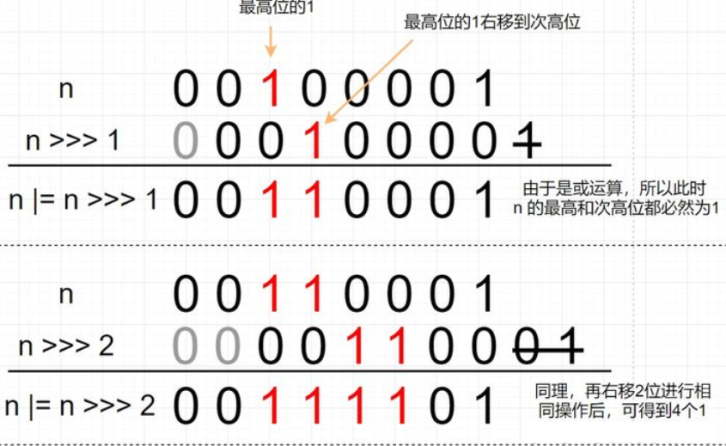
public Hashap(int initialCapacity,float loadFactor){  
 //找到>=initialCapacity的最小2^n  
 int capacity = 1;  
 while(capacity < initialCapacity){  
 //左移n位相当于乘以2的n次方  
 capacity <<= 1;   
 }  
}

在jdk1.8中

static final int tableSizeFor(int cap) {  
 int n = cap - 1;  
 n |= n >>> 1;  
 n |= n >>> 2;  
 n |= n >>> 4;  
 n |= n >>> 8;  
 n |= n >>> 16;  
 //经过这一连串移位运算后，会导致二进制下的n从最高位开始到最后全是1，最后结果再加上1，就是>=cap的最小2^n  
 return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;  
}

jdk8的源码比较难理解，但是也是可以理解的，首先我们跳过n=cap-1，看后面几句有什么作用：

假设 n 的值为 0010 0001，则该计算如下图：



我们可以清楚地看到：

// 参数值 -1 是为了处理传入参数值恰好是 2 的次幂的情况

int n = cap - 1;

//保证最高位和次高位都是1，即2个1  
 n |= n >>> 1;

//保证从最高位开始往低位数，连续4个1  
 n |= n >>> 2;

//同理，保证从最高位开始往低位数，连续8个1  
 n |= n >>> 4;

//同理，保证从最高位开始往低位数，连续16个1  
 n |= n >>> 8;

//同理，保证从最高位开始往低位数，连续32个1

//而java中的int一共就32位，也就是说不管你n有多大，经过这几步操作以后，保证n从最高位到最低位全是1  
 n |= n >>> 16;

//经过这一连串移位运算后，会导致二进制下的n从最高位开始到最后全是1，最后结果再加上1，就是>=cap的最小2^n  
 return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

计算机底层是二进制的，移位和或运算是非常快的，所以这个方法的效率很高，所以是一个非常巧妙的设计

* + - * 1. hash碰撞时头插改尾插

发生hash碰撞时，java 1.7 会在链表的头部插入，而java 1.8会在链表的尾部插入。改成尾插是为了避免在并发扩容(resize)时产生死循环问题

那么是不是就可以把hashmap用在多线程中呢？

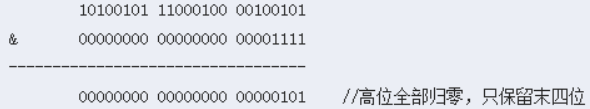
不行，因为put/get方法都没有加同步锁，有可能你上一秒put的值，下一秒想get用的时候已经被其他线程修改了，所以线程安全还是无法保证。

* + - * 1. 优化了扰动函数hash

hash函数可以用来计算key的hashcode，为什么不直接使用key的hashcode，而是要对hashcode进行hash计算呢？

这里有一个扰动的概念，hash函数实际上是扰动函数，如果不使用扰动函数，直接将key.hashCode()和下面的步骤2做与运算，则会有以下情景

以初始长度16为例，16-1=15。2进制表示是00000000 00000000 00001111。和某未扰动的hashcode做“与”操作如下(试图计算key在数组上的位置)：



很明显，高位完全没有起到作用，仅末4位起到了作用，这样加大了冲突的可能性，因此必须使用hash()作为扰动函数，不能直接使用hashcode

那么我们看一下java7和8的扰动函数的差别

java7：

static final int hash(int h) {  
 h ^= k.hashCode();  
 h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
}

Java8：

static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
}

这里可以看到，java8是采用了让**key.hashCode()**的低16位和高16位进行异或(^)

让高16位参与hash计算，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

明显java8简化了扰动函数，从java7的9次扰动变成了现在的2次扰动

* + - * 1. 修改了扩容时计算节点在新表的索引位置的方式

hashmap在resize(扩容)的过程,需要创建一个新的bucket数组,然后要将旧的bucket数组中的数据rehash到新的数组中

在java7中，旧map元素通过 h & (length-1)计算出该key在新bucket数组中的位置

而在java8中，这一计算方式改为了hash & oldCap：

next = e.next;  
if ((e.hash & oldCap) == 0) {  
 if (loTail == null)  
 loHead = e;  
 else  
 loTail.next = e;  
 loTail = e;  
}  
else {  
 if (hiTail == null)  
 hiHead = e;  
 else  
 hiTail.next = e;  
 hiTail = e;  
}

那么为什么会使用hash & oldCap代替h & (length-1)计算扩容时key在新bucket数组的索引位置呢？

那么假设现在hashmap的bucket数组大小为16，现有节点a（hash(key\_a)=5），节点b(hash(key\_b)=21)，那么他们在bucket数组中的index位置都是5，如下图：



那么现在hashmap要扩容了，扩容后bucket数组长度为32。我们发现，新表的length-1和旧表的length-1只比高位多了个1(红色标出)，如图：



计算a、b在新数组中index位置时，其他二进制位结果一致，仅取决于length-1新数组比旧数组多出来的这个高位1(即上图标红的1)，如果a、b节点的hash值与这个高位1的&运算结果一致，那么就表示a、b在新表中的位置也相同。而我们惊讶地发现，这个高位1（00010000），转成10进制正好是oldcap，即老数组长度，那么就是如下两种情况：

1. (e.hash & oldCap) == 0 ，则表示该key在新数组中还是原来的index位置
2. (e.hash & oldCap) != 0，则表示该key在新数组中的位置为“原index索引 + oldCap”。

这个设计确实非常的巧妙，帮助hashmap省去了在扩容时重新计算 hash 值的时间

* + - 1. fail-fast快速失败机制和fail-safe安全失败机制
    1. ConcurrentHashMap
       1. JDK1.7中的ConcurrentHashMap

在多线程情况下java7的HashMap时扩容会导致链表出现环形，这可能导致get操作出现死循环，这一切都是因为HashMap是线程不安全的，在jdk1.5以前，我们只能使用古老的HashTable来解决并发问题，HashTable中的方法基本上都是采用synchronized进行并发控制(或者使用Collections.synchronizedMap()，但它也是使用synchronized进行同步)，但synchronized的独占特性迫使多线程串行化访问容器，在锁竞争激烈的情形下这么做似乎没什么问题，但在竞争不激烈的情况下，这么做将严重影响集合的性能。因此从JDK1.5起，java提供了一个并发集合来解决HashMap的并发问题，那就是ConcurrentHashMap。它利用锁分段的思想，ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的Hashtable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行，大大提高了并发度

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁

**需要注意的是，不同于hashmap，ConcurrentHashMap的key和value都是不允许为null的（HashMap的key和value都可以为null）**

* + - * 1. java7中的ConcurrentHashMap

整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表”部分“或”一段“的意思，所以很多人都会将其描述为分段锁。

ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁住的是一个 Segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全。

public class ConcurrentHashMap<K, V> extends AbstractMap<K, V> implements ConcurrentMap<K, V>, Serializable {  
 //默认初始容量 16 ，是key-value的默认初始容量  
 static final int *DEFAULT\_INITIAL\_CAPACITY* = 16;  
 //默认加载因子 0.75  
 static final float *DEFAULT\_LOAD\_FACTOR* = 0.75f;  
 //默认的并发度 16 ->segments数组的大小   
 static final int *DEFAULT\_CONCURRENCY\_LEVEL* = 16;  
 //segments数组的大小的上限  
 static final int *MAXIMUM\_CAPACITY* = 1 << 30;  
 //Segment中HashEntry数组，也即是bucket数组的最小值  
 static final int *MIN\_SEGMENT\_TABLE\_CAPACITY* = 2;  
 //Segment的最大值，最大的并发度  
 static final int *MAX\_SEGMENTS* = 1 << 16; // slightly conservative  
 //锁重入的次数  
 static final int *RETRIES\_BEFORE\_LOCK* = 2;  
 //主要作用于keyhash过程  
 final int segmentMask;  
 final int segmentShift;  
 //存储数据节点的位置。Segment的数组  
 final Segment<K,V>[] segments;  
  
 //获取key对应集合  
 transient Set<K> keySet;  
 //获取key-value键值对集合  
 transient Set<Map.Entry<K,V>> entrySet;  
 //获取value的集合  
 transient Collection<V> values;  
  
//segment的锁是ReentantLock锁的子类 ，每个segment都有一个table数组   
static final class Segment<K,V> extends ReentrantLock implements Serializable {  
 static final int *MAX\_SCAN\_RETRIES* =  
 Runtime.*getRuntime*().availableProcessors() > 1 ? 64 : 1;  
  
 //存放数据  
 transient volatile HashEntry<K,V>[] table;  
 //存放数据的个数  
 transient int count;  
 transient int modCount;  
 transient int threshold;  
 final float loadFactor;  
}  
  
//HashEntry的结构  
static final class HashEntry<K,V> {  
 final int hash;  
 final K key;  
 volatile V value;  
 volatile HashEntry<K,V> next;  
}

}

ConcurrentHashMap 类中包含两个静态内部类 HashEntry 和 Segment。在分析ConcurrentHashMap类之前，我们先来看看这两个静态内部类

* + - * 1. 静态内部类HashEntry

HashEntry是ConcurrentHashMap类中用来封装映射表的键 / 值对的一个类，它的作用相当于HashMap中的Entry，但二者有一些不同的地方。

我们先来看看HashMap中的Entry是什么样的

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;  
 V value;  
 Entry<K,V> next;  
 int hash;

……

}

下面是HashEntry的源码，对比一下Entry类我们会发现二者极其地相似。不同的是，为了保证线程安全，我们在HashEntry里又看到了我们熟悉的类——Unsafe类：

static final class HashEntry<K,V> {

final int hash;  
 final K key;  
 volatile V value;  
 volatile HashEntry<K,V> next;

// 为了使用原子操作引入了Unsafe类   
 static final sun.misc.Unsafe *UNSAFE*;  
 static final long *nextOffset*;  
 static {  
 try {  
 *UNSAFE* = sun.misc.Unsafe.*getUnsafe*();  
 Class k = HashEntry.class;  
 *nextOffset* = *UNSAFE*.objectFieldOffset  
 (k.getDeclaredField("next"));  
 } catch (Exception e) {  
 throw new Error(e)s;  
 }  
 }  
  
 HashEntry(int hash, K key, V value, HashEntry<K,V> next) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = next;  
 }  
final void setNext(HashEntry<K,V> n) {  
 *UNSAFE*.putOrderedObject(this, *nextOffset*, n);  
 }  
}

* + - * 1. 静态内部类Segment

Segment 用来充当分段锁的角色，每个 Segment 对象守护散列映射表的部分HashEntry，因此Segment类提供了一个HashEntry<K,V>类型的数组——table ，可以把每个segment都看做一个小HashMap

static final class Segment<K,V> extends ReentrantLock implements Serializable {

private static final long *serialVersionUID* = 2249069246763182397L;

//操作失败后自旋的最大尝试次数

static final int *MAX\_SCAN\_RETRIES* = Runtime.*getRuntime*().availableProcessors() > 1 ? 64 : 1;

//HashEntry数组，在1.7的实现中，类似于1.7 HashMap中的bucket数组,当发生hash碰撞时，碰撞的HashEntry链接成一个链表，表头放在这个数组发生碰撞的bucket元素位上，每个table代表一个Segment，守护ConcurrentHashMap中的一部分bucket，一个ConcurrentHashMap默认有16个Segment，也即16个table，如果其中某一个Segment上发生了并发操作，锁住该Segament，不会影响另外15个Segment，table可扩容

transient volatile HashEntry<K,V>[] table;

//本Segment中包含的HashEntry元素个数

transient int count;

//table被更新的次数

transient int modCount;

//扩容的阈值，当table中的HashEntry元素数量超过threshold，Hash碰撞的概率大幅提升，此时需要扩容

transient int threshold;

//负载因子,和HashMap一样默认为0.75f

final float loadFactor;

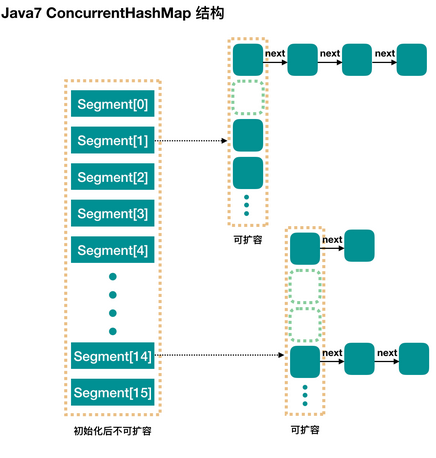
Segment(float lf, int threshold, HashEntry<K,V>[] tab) {  
 this.loadFactor = lf;  
 this.threshold = threshold;  
 this.table = tab;

}

……

}

* + - * 1. ConcurrentHashMap的构造



上图中，可扩容标记的是segment类中的table属性

public ConcurrentHashMap(int initialCapacity,float loadFactor, int concurrencyLevel) {

// 检查参数

if (!(loadFactor > 0) || initialCapacity < 0 || concurrencyLevel <= 0)  
 throw new IllegalArgumentException();

// 并发等级不能高于最大允许的Segment个数  
 if (concurrencyLevel > *MAX\_SEGMENTS*)  
 concurrencyLevel = *MAX\_SEGMENTS*;  
 // sshift是ssize从数字1开始向左位移的次数，当ssize大小为1时，sshift只能为0，当ssize大小为2时，sshift=1，当ssize=4时，sshift=2，当ssize=8时，sshift=3，当ssize=16时，sshift=4  
 int sshift = 0;  
 // ssize是Segment数组的大小

int ssize = 1;  
 // 以默认值concurrencyLevel=16计算，sshift=4，ssize=16

while (ssize < concurrencyLevel) {  
 ++sshift;  
 //ssize必须是2的倍数

ssize <<= 1;  
 }  
 this.segmentShift = 32 - sshift;  
 this.segmentMask = ssize - 1;

// initialCapacity不能大于最大容量  
 if (initialCapacity > *MAXIMUM\_CAPACITY*)  
 initialCapacity = *MAXIMUM\_CAPACITY*;  
 // c是每个Segment中HashEntry数组的大小，但c可能不是2的次幂

int c = initialCapacity / ssize;  
 if (c \* ssize < initialCapacity)  
 ++c;  
 //cap是每个Segment中的HashEntry数组的容量大小，经过while操作后，cap保证是2的次幂

int cap = *MIN\_SEGMENT\_TABLE\_CAPACITY*;  
 while (cap < c)  
 cap <<= 1;  
 //懒加载，只创建一个Segment[0]，剩下的等到用到再加载  
 Segment<K,V> s0 = new Segment<K,V>(loadFactor, (int)(cap \* loadFactor), (HashEntry<K,V>[])new HashEntry[cap]);  
 Segment<K,V>[] ss = (Segment<K,V>[])new Segment[ssize];  
 *UNSAFE*.putOrderedObject(ss, *SBASE*, s0);   
 this.segments = ss;  
}

* + - * 1. 并发扩容
        2. put方法

public V put(K key, V value) {

Segment<K,V> s;  
 if (value == null)  
 throw new NullPointerException();  
 //计算hash值

int hash = hash(key);

//根据hash值找到key应当在Segment数组中的哪一个位置，即该key应放在索引为j的Segment中  
 int j = (hash >>> segmentShift) & segmentMask;

//因为构造函数是懒加载，所以这里要判断Segment是否存在  
 if ((s = (Segment<K,V>)*UNSAFE*.getObject(segments, (j << *SSHIFT*) + *SBASE*)) == null)   
 s = ensureSegment(j);

//如果存在就把key-value放入s这个Segment中，这是一个加锁操作  
 return s.put(key, hash, value, false);  
}

* + - * 1. get方法

public V get(Object key) {

Segment<K,V> s;   
 HashEntry<K,V>[] tab;

//计算hash值  
 int h = hash(key);  
 long u = (((h >>> segmentShift) & segmentMask) << *SSHIFT*) + *SBASE*;  
 //根据hash值找到对应的Segment——s

if ((s = (Segment<K,V>)*UNSAFE*.getObjectVolatile(segments, u)) != null &&  
 (tab = s.table) != null) {

//找到Segment内部的数组链表，遍历  
 for (HashEntry<K,V> e = (HashEntry<K,V>) ***UNSAFE***.getObjectVolatile  
 (tab, ((long)(((tab.length - 1) & h)) << *TSHIFT*) + *TBASE*);  
 e != null; e = e.next) {  
 K k;  
 if ((k = e.key) == key || (e.hash == h && key.equals(k)))  
 return e.value;  
 }  
 }  
 return null;  
}

* + - * 1. size()和containsValue()的实现
      1. JAVA 8中的ConcurrentHashMap
         1. java7实现方式的缺陷

JDK1.7的设计固然精妙，但是存在一定缺陷；即根据key寻找value时，都需要经过两步：

1. **确认key属于哪个segment**
2. **确认key位于数组+链表的哪一个位置**

这一点可以在1.7的get方法的源码中得到确认

* + - * 1. java8实现方式

相比于jdk7，java8对ConcurrentHashMap做了如下改动：

1. 进一步降低了锁的粒度，JDK1.7 版本锁的粒度是基于 Segment 的，包含多个节点（HashEntry），而 JDK1.8 锁的粒度就是单节点（Node）
2. 锁的粒度进一步降低，segment已经无用，但为了兼容还是保留了
3. 通hashMap一样，采用node数组+链表（红黑树）的结构提高hash冲突链表变长引发的查找性能下降
4. 通过synchronized关键字+cas取代了原先的ReentrantLock
   * 1. HashTable

HashTable很少使用，基本都用ConcurrentHashMap

1. HashMap没有考虑同步，是线程不安全的；Hashtable使用了synchronized关键字，是线程安全的；
2. HashMap允许K/V都为null；Hashtable后者K/V都不允许为null；
3. HashMap默认容量为16，每次扩容都是原来的两倍，HashTable为11，每次扩容都是原来的2n+1
4. HashMap继承自AbstractMap类；而Hashtable继承自Dictionary类；
   * 1. LinkedHashMap

hashmap不存在保存顺序的机制，而LinkedHashMap中可以保持两种顺序，分别是**插入顺序**和**访问顺序**，这个是可以在LinkedHashMap的初始化方法中进行指定的。

LinkedHashMap是HashMap的亲儿子，直接继承HashMap类。然后自定义了一个静态内部类Entry<K,V>继承自HashMap.Node<K,V>。

static class Entry<K,V> extends HashMap.Node<K,V> {  
 Entry<K,V> before, after;  
 Entry(int hash, K key, V value, Node<K,V> next) {  
 super(hash, key, value, next);  
 }  
}

然后LinkedHashMap重写了父类HashMap中的newNode()方法，保证调用时创建的都是LinkedHashMap自己的Entry

Node<K,V> newNode(int hash, K key, V value, Node<K,V> e) {  
 LinkedHashMap.Entry<K,V> p =  
 new LinkedHashMap.Entry<K,V>(hash, key, value, e);  
 linkNodeLast(p);  
 return p;  
}

private void linkNodeLast(LinkedHashMap.Entry<K,V> p) {  
 LinkedHashMap.Entry<K,V> last = tail;  
 tail = p;  
 if (last == null)  
 head = p;  
 else {  
 p.before = last;  
 last.after = p;  
 }  
}

很显然，这里把newNode创建的节点记录到**LinkedHashMap.Entry<K,V>**双向链表中了，这个双向链表可以用来记录顺序，这个顺序可以是插入顺序也可以是访问顺序

其他还有一些不同不细说了，大致上和HashMap差不太多，毕竟是子类

* + 1. TreeMap

线程不安全的有序map，底层是红黑树实现。

可以指定比较器（Comparator 比较器），通过重写 compare 方法来自定义排序；如果没有指定比较器，TreeMap 默认是按 Key 的升序排序（如果 key 没有实现 Comparable接口，则会抛异常）

* 1. Set
     1. HashSet

答：HashSet的底层其实就是HashMap，只不过我们HashSet是实现了Set接口并且把数据作为K值，而V值一直使用一个相同的虚值来保存，我们可以看到源码：

public boolean add(E e) {

return map.put(e, PRESENT)==null;// 调用HashMap的put方法,PRESENT是一个至始至终都相同的虚值

}

由于HashMap的K值本身就不允许重复，并且在HashMap中如果K/V相同时，会用新的V覆盖掉旧的V，然后返回旧的V，那么在HashSet中执行这一句话始终会返回一个false，导致插入失败，这样就保证了数据的不可重复性；



* 1. 快速失败机制fail-fast

如果存在两个线程A、B，线程A正在对集合对象使用迭代器遍历，线程B对集合进行更改(新增、删除、修改)，都会导致触发fail-fast机制，使得A线程抛出ConcurrentModificationException异常

其原理是，迭代器会保存一个叫modCount的变量，并且在迭代器调用hasNext()/next()方法前都会先检查modCount是否被改变，如果被改变了，就抛出异常。另一边，如果有另外一个线程执行更改(新增、删除、修改)操作，都会去修改modCount的值，这样就实现了fail-fast机制

Java.util包下的集合都是快速失败的

* 1. 安全失败机制fail-safe

采用安全失败机制fail-safe的集合，在遍历时并不是直接在集合内容上进行访问的，而是先复制原有集合内容，在副本上进行遍历，因此遍历过程中对集合的改动并不能被迭代器检测到，就不会抛出异常，典型的如CopyOnWriteArrayList

Java.util.concurrent包下的集合都是快速失败的

1. JAVA 反射

## 5.1 反射概述

反射 (Reflection) 是 Java 的特征之一，它允许运行中的 Java 程序获取自身的信息，并且可以操作类或对象的内部属性。通过反射，我们可以在运行时获得程序或程序集中每一个类型的成员和成员的信息。程序中一般的对象的类型都是在编译期就确定下来的，而 Java 反射机制可以动态地创建对象并调用其属性，这样的对象的类型在编译期是未知的。所以我们可以通过反射机制直接创建对象，即使这个对象的类型在编译期是未知的。

Java的反射机制是通过**Class对象**来实现的，在Java中，Object 类是所有类的根类，而Class类则是描述Java类的类。在Java中，每一个类都有一个相应的**Class对象**，在将Java源码编译成.class文件后就会生成一个**Class对象**，并将**Class对象**保存在.class文件中。

反射的优点：

1. **灵活性高，可拓展性强，只有到运行时才动态创建获取对象实例**

反射的缺点：

1. **执行效率低下，影响性能。反射需要动态的解析类的信息，相比于非反射使用的方式要慢**
2. **暴露了内部的封装，可能会引起一些负面效果。比如不该被外部调用的私有方法，通过反射被调用了**

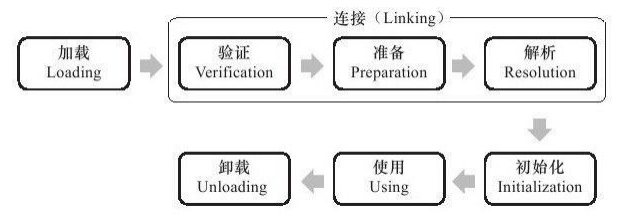
## 5.2 类的加载机制

**.class文件**由**类装载器(ClassLoader)**装载后，在JVM中将生成一个**Class对象**，通过**Class对象**可以获知类的结构信息：如构造函数，属性和方法等

类的加载是将类的**.class文件**中的二进制数据读入到内存中，将其放在运行时数据区的**方法区**内，然后在堆区创建一**个java.lang.Class对象**，用来封装类在方法区内的数据结构。类的加载，最终会获得一个**Class对象**。

### 5.2.1机制原理

**一个**类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段



#### 5.2.1.1 加载

在类的加载阶段，虚拟机需要做如下三件事情：

1. **通过一个类的全限定名来获取定义这个类的二进制字节流（也就是先通过路径找到这个类的.class文件）**
2. **将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构**
3. **在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口**

获取二进制字节流的方式非常多，并不一定要从.class文件获取，只要一段二进制字节流符合Class文件的规范，都可以当做一个Class文件，例如我们可以从jar包中获取，从网络中获取，运行时生成等

加载步骤是由**类加载器**完成的，可以使用默认的类加载器，也可以使用自定义类加载器。加载阶段完成后，类的二进制字节流就按照虚拟机所需的格式存储在方法区之中，而且在Java堆中也创建了一个**java.lang.Class对象**

#### 5.2.1.2 验证

验证是连接阶段的第一步，这一阶段的目的是**为了确保.class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全**，**验证阶段有可能触发更多class的加载**

**验证阶段是非常重要的，但不是必须的**。它对程序运行期没有影响，如果所引用的类经过反复验证，那么可以考虑采用-Xverifynone参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

此阶段主要包含如下几个部分的验证：  
**1.文件格式验证**

验证字节流是否符合.class文件格式的规范，并且能被当前版本的虚拟机处理。

**2.元数据验证**

对字节流进行语义分析，以保证其描述的信息符合Java语法的要求，如：

**①这个类是否有父类。（除了java.lang.Object之外）**

**②这个类的父类是否继承了不允许的类。（final修饰的类）**

**③如果这个类不是抽象类，是否实现了父类或接口中要求实现的所有方法。**

**④类中的字段，方法是否与父类产生矛盾等**

**3.字节码验证**

进行数据流和控制流分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的行为，如：

**①保证跳转指令不会跳转到方法体以外的字节码指令上**

**②保证方法体中的类型转换是有效的等等。**

**4.符号引用验证**

这个阶段发生在虚拟机将符号引用转化为直接引用时候**（解析阶段）**，确保解析能够正常的执行。如：

**①符号引用中通过字符串描述的全限定名是否能找到对应的类。   
②在指定类中是否存在符合方法和字段描述所对应的方法和字段。   
③符号引用中的类、字段、方法的访问性（private、protected、public、default）等等**

#### 5.2.1.3 准备

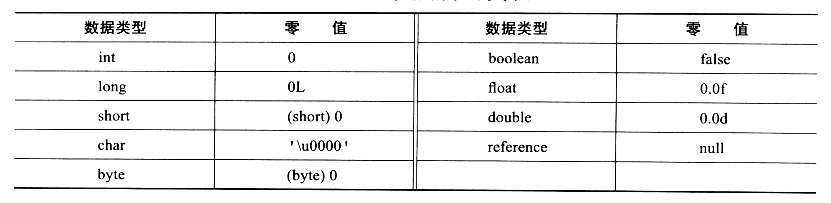
准备阶段主要负责**为类的静态变量(static)分配内存(在方法区中分配)，并将其初始化为默认值**

需要注意的是：

**1.这里的初始化是指将数据类型初始化为默认的零值（如0、0L、null、false等），而不是被在Java代码中被显式地赋予的值，final属性除外，因为被final修饰的变量，会在编译器就将值放入常量池中**

如：public static int value = 3;在准备阶段结束后，value将被初始化为0而不是3

public **final** static int value = 3;在准备阶段结束后，value将被初始化为3



**2.这时候进行内存分配(static类变量在方法区中分配)的仅包括类的静态变量(static声明的类属性)，而不包括实例变量(非static声明的类属性)，实例变量会在类实例化对象的时候随着对象一块分配在Java堆中。**

如：

public int key= 1;

public static int value = 3;

key在这一阶段不会分配内存，要等到**类被使用时，即实例化对象时**才会分配内存。

value将会在方法区中分配内存

#### 5.2.1.4 解析

**解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程**

那么什么是符号引用，什么是直接引用呢？

**符号引用：**符号引用以一组符号来描述所引用的目标，符号可以是任意形式的字面量，只要使用时能无歧义的定位到目标即可。符号与虚拟机实现的内存布局无关，所引用的目标并不一定加载到内存中（方法区）

**直接引用：**直接引用可以**指向目标的指针**、**相对偏移量**或者是一个**能够直接定位到目标的句柄**。直接引用于虚拟机的内存布局相关，同一个符号引用在不同的虚拟机实例上翻译出来的直接引用一般不同。如果有了直接引用，那么，所引用的目标一定已经在内存中存在

 说白了，符号引用只是一组用于描述被引用目标的符号，并不能直接被虚拟机使用，并不直接指向任何内存地址，而直接引用指向的目标必须真实存在于内存之中的。**在代码运行过程中，会不断产生新对象，故而解析这一过程并不是一次就完成的，其发生的时机不固定。**

#### 5.2.1.5 初始化

**类的初始化的主要工作是对类的静态变量，静态代码块执行初始化操作**

在准备阶段，我们已经为所有**非final的静态变量**分配了内存地址，现在，我们需要对其进行初始化操作

JVM并不需要对所有加载链接过的类进行初始化操作，只有当如下几种情形发生时，JVM才会对其进行初始化：

**1）创建类的实例，也就是new一个对象时**

**2）访问某个类或接口的静态变量，或者对该静态变量赋值时**

**3）调用类的静态方法时**

**4）反射（Class.forName("com.lyj.load")）时**

**5）初始化一个类的子类（会首先初始化子类的父类）时**

**6）JVM启动时标明的启动类(如包含main的类)**

JVM初始化的步骤如下：

**1）如果这个类还没有被加载和链接，那先进行加载和链接**

**2）假如这个类存在直接父类，并且这个类还没有被初始化（注意：在一个类加载器中，类只能初始化一次），将先行初始化直接父类（不适用于接口）**

**3 ) 假如类中存在初始化语句（如static变量和static块），那就依次执行这些初始化语句。**

**注意：**

**1.static变量和static块之间的执行顺序由代码编写顺序决定**

**2.子类访问父类的静态变量时，父类的静态变量会初始化，而子类的静态变量不会**

**3.通过数组定义来引用类，不会触发类的初始化**

**4.访问类的常量(final)，不会触发初始化，因为常量在类加载时就已经被赋值了**

**至此，类加载结束，但是请留意，这里还没有涉及到类的实例化，也就是说此时还没有开始new操作。一个类，必须经过 加载-链接-初始化 后才允许被实例化**

#### 5.2.1.6例子

**例1：**

class Father {

static {  
 System.*out*.println("执行父类静态代码块:father=" + Father.*father* + " finalFather=" + Father.*finalFather*);  
 }  
 {  
 System.*out*.println("执行父类代码块");  
 }  
 public final static int *finalFather* = 30;  
 public static int *father* = 20;  
 public Father() {  
 System.*out*.println("执行父类构造方法");  
 }  
}  
  
class Son extends Father {  
 public static int *son* = 10;  
 static {  
 System.*out*.println("执行静态代码块:son=" + *son*);  
 }  
 {  
 System.*out*.println("执行子类代码块");  
 }  
 public Son() {  
 System.*out*.println("执行子类构造方法");  
 }  
}

执行结果为：

//执行父类静态代码块:father=0 finalFather=30

//执行静态代码块:son=10  
//执行父类代码块  
//执行父类构造方法  
//执行代码块  
//执行构造方法

**父类的（静态变量、静态初始化块）=>**

**子类的（静态变量、静态初始化块）=>**

**父类的（变量、初始化块、构造器）=>**

**子类的（变量、初始化块、构造器）**

**例2：**

public class Leetcode {

public static void main(String[] args) {  
 System.*out*.println(SonClass.*value*); // 访问静态变量  
 SonClass[] sca = new SonClass[10]; // 访问数组引用  
 }  
}  
  
class SuperClass {  
 static {  
 System.*out*.println("superclass init");  
 }  
 public static int *value* = 123;  
}  
  
class SonClass extends SuperClass {  
 static {  
 System.*out*.println("sonclass init");  
 }  
}

执行结果为：

// superclass init

// 123

**子类访问父类的静态变量时，父类会初始化，而子类不会**

**访问数组引用不会触发初始化**

**例3：**

public class Leetcode {

public static void main(String[] args) {  
 System.*out*.println(ConstClass.*HELLOWORLD*);// 调用类常量  
 }  
}  
class ConstClass {  
 static {  
 System.*out*.println("ConstClass init");  
 }  
 public static final String *HELLOWORLD* = "hello world";  
}

执行结果为：

// hello world

**访问final静态变量不会触发初始化**

**例4:**

public class Leetcode {

public static void main(String[] args) {  
 SingleTon singleTon = SingleTon.*getInstance*();  
 System.*out*.println("count1=" + singleTon.*count1*);  
 System.*out*.println("count2=" + singleTon.*count2*);  
 }  
}  
  
class SingleTon {  
 private static SingleTon *singleTon* = new SingleTon();  
 public static int *count1*;  
 public static int *count2* = 0;  
  
 private SingleTon() {  
 *count1*++;  
 *count2*++;  
 }  
  
 public static SingleTon getInstance() {  
 return *singleTon*;  
 }  
}

执行结果为：

// count1=1

// count2=0

**分析：**

1. **首先调用静态方法SingleTon.*getInstance*()，触发了*****SingleTon*类的初始化，但此时发现，*SingleTon*类并未被加载，因此先进行加载操作**
2. **读取.class文件，并为*SingleTon*类的静态变量分配内存空间，因此：**

**singleTon = null**

**count1 = 0**

**count2 = 0**

1. **加载阶段结束后，紧接着进行链接，链接结束后进行*SingleTon*类的初始化，singleTon被赋值为new SingleTon()。此时执行构造函数，**

**singleTon = new SingleTon()**

**count1 = 1**

**count2 = 1**

1. **接着初始化count1和count2**

**singleTon = new SingleTon()**

**count1 = 1**

**count2 = 0**

### 5.2.2类加载器 ClassLoader

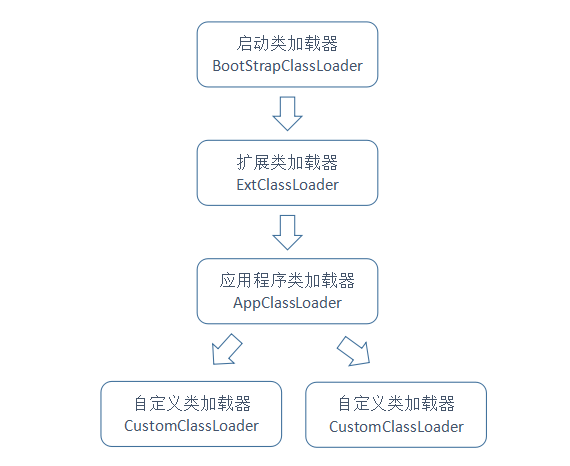
对于任意一个Java类，都**必须通过类加载器从.class文件加载到JVM的方法区，并在JVM堆中生成java.lang.Class对象**才能使用类的各个功能，Classloader只负责”加载-链接-初始化”中的加载这一步。ClassLoader并不是将所有的.class文件一次加载到JVM中的，而是按需加载，用到了就加载

ClassLoader被用来加载.class文件，它负责将 .Class 的字节码转换成内存形式的 java.lang.Class 对象存储在堆中，字节码可以来自jar包，也可以来自网络等处

Java为我们提供了三种内置的类加载器BootStrapClassLoader、ExtClassLoader、AppClassLoader

其中BootStrapClassLoader由c++实现，没有父类，而AppClassLoader继承自ExtClassLoader，ExtClassLoader继承自BootStrapClassLoader。而其他类加载器(包括ExtClassLoader和AppClassLoader)本质上也是由java实现的java类，因此也需要类加载器对其进行加载。因此，

**拓展类加载器(ExtClassLoader)和系统类加载器(AppClassLoader)都是由启动类加载器(BootStrapClassLoader)加载的。**



#### 5.2.2.1 BootStrapClassLoader启动类加载器

**负责加载jdk中的核心类库**，如rt.jar、resources.jar、charsets.jar等。该加载器由c++实现，并不继承java.lang.ClassLoader类，因此我们无法在自己的程序中显示获取BootStrapClassLoader类的实例对象

**除了引导类加载器之外，所有的类加载器都有一个父类加载器，这里的父类子类不是通过传统的继承实现的，而是通过一种“包含”关系(每一个子类都有一个属性用于存放父类加载器的引用)实现的。因此，下面的加载器中的“继承”都是带双引号的。**

#### 5.2.2.2 ExtClassLoader扩展类加载器

**负责加载 Java 的扩展库。**Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。由java编写，因此我们可以在自己的程序中使用它。由于其父类启动类加载器由c++实现，扩展类加载器通过getParent()方法获取的父类加载器就是null

**扩展类加载器(ExtClassLoader)“继承”自启动类加载器(BootStrapClassLoader)**

**扩展类加载器(ExtClassLoader)由启动类加载器(BootStrapClassLoader)负责加载**

#### 5.2.2.3 AppClassLoader系统类加载器

**负责加载用户类路径(ClassPath)上所指定的类库。**开发者可直接使用该类加载器，可以通过ClassLoader.getSystemClassLoader()来获取它。一般来说，开发者自定义的类就是由应用程序类加载器加载的。

**系统类加载器(AppClassLoader)“继承”自扩展类加载器(ExtClassLoader)**

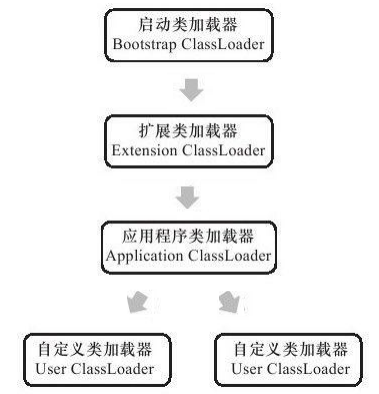
除了上述加载器外，我们还可以自己定制自己的类加载器，只需要我们继承ClassLoader类即可。**一般来说，开发人员编写的类加载器的父类加载器是系统类加载器(AppClassLoader)**

### 5.2.3双亲委派(Parents delegate)

双亲委派(Parents delegate)是一种翻译上的错误，所谓的双亲委派(Parents delegate)，实际上应当翻译为“父辈委托”。

上面也说了，这里的父辈并不是父子继承的父辈，而是“子类”中存放了“父类”的引用，并非继承，而是包含的关系。Parents加了个s，表示的是父辈这一群体，并不是表示双亲的意思，意思是父辈加载不了就委托子辈进行

双亲委派模型：如果一个类加载器收到了一个类的加载请求时，它不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层级的类加载器都是如此，这样一级级传递上去，直到顶层的启动类加载器(BootStrapClassLoader)，在此期间并没有任何一个加载器进行了实质上的加载。也就是说，**加载是从启动类加载器(BootStrapClassLoader) 开始，自上而下，如果父辈加载器无法加载，就委托给子加载器进行加载，如果最终无法找到对应加载器，就抛出ClassNotFoundException异常**。



好处：

1. **java类随着它的类加载器一起具备了一种带有优先级的层次关系**。例如类java.lang.Object，它存放在rt.jar中，无论哪个类加载器要加载这个类，最终都会先委派给启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类(在java中，即使是同一个类的.class文件，被不同的类加载器加载后，JVM也认为它们是不同的类)。相反，如果用户自己写了一个名为java.lang.Object的类，并放在程序的Classpath中，那系统中将会出现多个不同的Object类，java类型体系中最基础的行为也无法保证，应用程序也会变得一片混乱。
2. 可以避免重复加载，保证同一个类只加载一次，当父辈已经加载了该类的时候就没有必要 ClassLoader再加载一次

### 5.2.4自定义类加载器(UserClassLoader)

在很多时候我们也需要自己定制类加载器来使用，以便满足我们的特殊要求

自定义加载类只要做两个步骤：

1. **定义类。继承ClassLoader**
2. **重写ClassLoader的 findClass方法**

ClassLoader类中有三个重要的方法：loadClass()、findClass() 和 defineClass()

//加载指定名称（包括包名）的二进制类型，供用户调用的接口。它首先会查找当前 ClassLoader 以及它的双亲里面是否已经

加载了目标类，如果没有找到就会让双亲尝试加载，如果双亲都加载不了，就会调用 findClass() 让ClassLoader自己来查找要加载的目标类

public Class<?> loadClass(String name);

//根据name查找对应的类，读取字节码，最后通过调用defineClass，将二进制数据转换为java.lang.Class 的实例并返回  
protected Class<?> findClass(String name)  
//定义类型，一般在findClass方法中读取到对应字节码后调用，不可继承（说明：JVM已经实现了对应的具体功能，解析对应的字节码，产生对应的内部数据结构放置到方法区，所以无需覆写，直接调用就可以了）  
protected final Class<?> defineClass(String name, byte[] b, int off, int len) throws ClassFormatError{}

**不要轻易覆盖 loadClass 方法，否则可能会破坏双亲委派规则，导致自定义加载器无法加载内置的核心类库。**

Eg：

class CustomClassLoader extends ClassLoader {

Class findClass(String name) {  
 // 查找二进制字节码,自定义方法  
 byte[] code = findCodeFromSomewhere(name);  
 // 组装java.lang.Class对象  
 return this.defineClass(code, name);  
 }  
}

有意思的是，**Java 虚拟机是如何判定两个 Java 类是相同的时，虚拟机不仅要看类的全限定名是否相同，还要看加载此类的类加载器是否一样。**只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。这一点，在我们开发自己的类加载器时十分重要

### 5.2.5 类加载的三种方式

我们通常使用三种方式加载.class文件中的类：new、Class.forName()、ClassLoader实例对象的loadClass()

#### 5.2.5.1 new关键字

和Class.forName()方法一样，new关键字将使用当前类的类加载器加载类，即this.getClass.getClassLoade。如果找不到类，该种方式将会抛出NoClassDefFoundError，这是一个Error而非异常，编译都不会通过

#### 5.2.5.2 ClassLoader实例对象的loadClass()

通过调用某个的ClassLoader实例对象的loadClass()方法加载类，和Class.forName()一样，在找不到要加载的类时，将抛出ClassNotFoundException异常，但在编译阶段无法监测出该错误，eg：

try {

java.lang.Class serviceClass=java.lang.Class.*forName*("com.test.service.impl.TestServiceImpl");  
 TestServiceImpl impl= (TestServiceImpl) serviceClass.newInstance();  
 impl.dd();  
} catch (ClassNotFoundException | IllegalAccessException | InstantiationException e) {  
 e.printStackTrace();  
}

#### 5.2.5.3 Class.forName()静态方法

Java.lang.Class类还提供了一个静态方法forName()来加载类，该方法将使用调用了forName()方法的那个类的ClassLoader对指定的类进行加载，eg：

下例中，就将调用**Leetcode**类的ClassLoader对**Hello**类进行加载

public class Leetcode {

public static void main(String[] args) {  
 try {  
 java.lang.Class<?> x = java.lang.Class.*forName*("Hello");  
 System.*out*.println(x);  
 x = ClassLoader.*getSystemClassLoader*().loadClass("Hello");  
 System.*out*.println(x);  
 } catch (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
 }  
}

当然，也提供了一个可以指定ClassLoader版本的forName()方法

public static Class<?> forName(String name, boolean initialize, ClassLoader loader)

**initialize**参数将决定类是否会被初始化(注意不是实例化)

Class.forName()和普通的类加载loadClass

1. **Class.forName返回的Class对象可以自行决定是否初始化。而ClassLoader.loadClass返回的类型绝对不会初始化，最多只会做连接操作。**
2. **Class.forName可以自行决定由哪个classLoader来请求这个类型。而ClassLoader.loadClass则是用当前的classLoader去请求**

## 5.3 反射机制

### 5.3.1 获取Class对象的几种方式

在JAVA中，每一个类都有一个相应的Class对象(也可能有多个，这是由多个类加载器加载出来的，一般来说只有一个)，我们通常通过如下几种方式获取Class对象：

1. **调用类本身的.class属性**

Class clazz = String.class;

**2.通过类的实例对象获取Class对象**

Person p = new Person();

Class clazz = p.getClass();

**3.通过Class类的静态方法forName**

Class clazz = Class.forName("java.util.commons");

**4.通过类加载器获取Class对象**

ClassLoader classLoader = this.getClass().getClassLoader();

Class clazz = classLoader.loadClass("java.util.commons");

**Eg:假设有班级类Classes**

public class Classes {

private String teacherName;  
 protected String[] students;

public int age;

public Classes(String teacherName) {  
 this.teacherName = teacherName;  
 }  
  
 public Classes(String teacherName, String[] students) {  
 this.teacherName = teacherName;  
 this.students = students;  
 }  
  
 public String getTeacherName() {  
 return teacherName;  
 }  
  
 public void setTeacherName(String teacherName){  
 this.teacherName=teacherName;  
 }  
}

### 5.3.2 Class类

Class对象是由类加载器加载.class文件所得。一般来说，一个类在被加载后会有一个Class对象与之对应，当然，如果有多个类加载器对类进行了加载，则会有多个Class对象，Class对象的常用方法如下：

//获取父类的class对象

public native Class<? super T> getSuperclass()

//获取类的 成员类 信息(public/继承)

public Class<?>[] getClasses()

//获取所有类的 成员类 信息(除继承的方法以外)

public Class<?>[] getDeclaredClasses()

1. 不带 "Declared"的方法支持取出包括继承、公有（Public） ， 但不包括有（Private）的构造函数
2. 带 "Declared"的方法是支持取出包括公共（Public）、保护（Protected）、默认（包）访问和私有（Private）的构造方法 ， 但不包括继承的构造函数

### 5.3.3 Constructor构造函数类

我们常常使用Class对象的newInstance()方法来实例化一个对象，但newInstance()方法只能使用无参构造函数实例化对象，这时候就需要获取Constructor构造类函数进行实例化。

在使用中，我们一般有两种方法获取Class对象的Constructor对象：

// 遍历所有构造函数，找出参数列表数据类型符合parameterTypes的构造函数(public/继承)

public Constructor<T> getConstructor(Class<?>... parameterTypes)

// 获取Class对象的所有构造函数(public/继承)

public Constructor<?>[] getConstructors()

// 遍历所有构造函数，找出参数列表数据类型符合parameterTypes的构造函数(除继承的构造函数以外)

public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)

// 获取Class对象的所有构造函数(除继承的构造函数以外)

public Constructor<?>[] getDeclaredConstructors()

eg：

Constructor constructor = Classes.class.getConstructor(String.class,String[].class);

Constructor[] constructors = Classes.class.getConstructors();

通过构造函数对象的newInstance方法实例化对象

Classes classes=(Classes)constructor.newInstance("老师",new String[]{});

1. 不带 "Declared"的方法支持取出包括继承、公有（Public） ， 但不包括有（Private）的构造函数
2. 带 "Declared"的方法是支持取出包括公共（Public）、保护（Protected）、默认（包）访问和私有（Private）的构造方法 ， 但不包括继承的构造函数

### 5.3.4 Method方法类

Method类的对象代表Class对象中的一个方法，我们通常使用以下方法获取Class对象的Method对象：

//根据 方法名和方法的参数列表数据类型 查询方法(public/继承)

public Method getMethod(String name, Class<?>... parameterTypes);

//根查询所有方法(public/继承)

public Method[] getMethods();

//根据 方法名和方法的参数列表数据类型 查询方法(除继承的方法以外)

public Method getDeclaredMethod(String name, Class<?>... parameterTypes);

//根查询所有方法(除继承的方法以外)

public Method[] getDeclaredMethods();

1. 不带 "Declared"的方法支持取出包括继承、公有（Public） ， 但不包括有（Private）的构造函数
2. 带 "Declared"的方法是支持取出包括公共（Public）、保护（Protected）、默认（包）访问和私有（Private）的构造方法 ， 但不包括继承的构造函数

### 5.3.5 Field属性类

Field类的对象代表Class对象中的一个属性，我们通常使用以下方法获取Class对象的Field对象：

//根据属性名查询属性(public/继承)

public Field getField(String name);

//查询所有属性(public/继承)

public Field[] getFields();

//根据属性名查询属性(除继承的属性以外)

public Field getDeclaredField(String name);

//根据属性名查询属性(除继承的属性以外)

public Field[] getDeclaredFields();

1. 不带 "Declared"的方法支持取出包括继承、公有（Public） ， 但不包括有（Private）的构造函数
2. 带 "Declared"的方法是支持取出包括公共（Public）、保护（Protected）、默认（包）访问和私有（Private）的构造方法 ， 但不包括继承的构造函数

Eg：

//根据属性名查询public属性

Field age =Classes.class.getField("age");  
//根据属性名查询属性(包括private)  
Field teacherName=Classes.class.getDeclaredField("teacherName");  
//查询所有public属性  
Field[] publicFields=Classes.class.getFields();  
//查询所有属性，包括private  
Field[] fields=Classes.class.getDeclaredFields();

通过反射机制中Field类的setAccessible()方法，我们还可以强行破坏类的封装性：

Classes classes=(Classes)constructor.newInstance("老师",new String[]{});

//根据属性名查询私有属性teacherName  
Field teacherName=Classes.class.getDeclaredField("teacherName");

//暴力访问，强行修改teacherName的访问性  
teacherName.setAccessible(true);

//访问private属性

System.*out*.println("classes.getTeacherName() = " + classes.getTeacherName());

//该方法受属性访问权限控制，只能修改public类型的属性，这里破坏了teacherName的封装性才允许修改  
teacherName.set(classes,"新老师");

//该方法受属性访问权限控制，只能获取public类型的属性，这里破坏了teacherName的封装性才允许读取

System.*out*.println("teacherName = " + teacherName.get(classes));

1. JAVA 多线程和并发
   1. 简介

在过去单 CPU 时代，单个CPU在一段运行时间内只能执行单一程序。之后发展到多任务阶段，计算机能在同一时间点并行执行多任务或多进程。虽然并不是真正意义上的“同一时间点”，而是多个任务或进程共享一个 CPU，并由操作系统来完成多个任务间的来回切换，以使得每个任务都有机会获得一定的时间片(极短的时间片)运行。

表面上看起来，因为不同任务之间的来回切换有一定的性能损耗，似乎顺序执行程序比多线程并行执行程序更高效。但在实际的编程中，由于阻塞的存在(通常是I/O)，使得一切变得不同了，一旦发生阻塞，顺序执行的程序就会停顿下来，整个CPU处于无事可做的状态，极度浪费资源。并发因此出现

### 6.1.1多进程

进程就是一个程序在计算机上的一次执行过程，是CPU上下文切换的程序执行时间总和，一个程序，当得到CPU的时候，相关的资源必须也已经就位，比如打印机，显示屏幕，GPS，用户输入等，必须立刻能够执行，它才有资格得到CPU的一次使用机会，当它得到CPU的时候，必须加载它上一次获得CPU执行后所保存的环境(如函数、变量等)，然后执行，分配给他的CPU执行时间用完了后，保存本次执行的环境(上下文)，切换出去，等待着下一次使用CPU，因此：

**一次CPU执行进程的时间**= **CPU加载上下文+CPU执行+CPU保存上下文**

### 6.1.2多线程

进程的颗粒度太大，每次都要有上下文的调入，加载，执行，保存，调出，它是可以再分割的。如果把打开QQ比作进程，那么传输文字开一个线程、传输语音开了一个线程、弹出对话框又开了一个线程。如果把打开迅雷比作进程，那么每打开一个下载任务就是执行一个线程。因此：

**多个线程组成了进程**

**当然，线程也是有上下文的，并且下面我们将多次提到线程上下文切换**

### 6.1.3并发和并行

实现并发最直接的方式是使用操作系统级别的进程，进程和进程之间很少互相干涉，即使他们使用的是同一台机器的内存，但他们并不共享内存(进程A结束会保存自己的上下文，进程B使用的也是自己的上下文)

JAVA使用多线程的方式实现并发，这就意味着线程之间是共享内存和IO的，也就是说，如果你在线程A中将变量var修改了值，它会影响到B线程的执行；线程A要读取文件F的同时，线程B在对文件F进行写入(这在进程实现的并发中不可能出现)，线程B势必影响线程A的读取，这无疑加大了编程的难度，但它更加灵活、强大

并行则是指多个cpu实例或者多台机器同时执行一段处理逻辑，是真正的同时，并不是通过上下文切换轮流使用cpu而模拟出的同时

### 6.1.4线程安全与同步

**线程安全**就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可使用。不会出现数据不一致或者数据污染。(Vector,HashTable等都是线程安全型集合)

**线程不安全**就是不提供数据访问保护，有可能出现多个线程先后更改数据导致所得到的数据是脏数据。（ArrayList，LinkedList，HashMap等都是线程不安全型集合）

**线程同步**是为了防止多个线程同时访问同一个数据对象时，对数据造成破坏，通过人为的控制和调度，保证共享资源的多线程访问的线程安全。通俗的说就是让线程们”排队”使用数据，不要蜂拥而上

### 6.1.5线程的生命周期

线程在其生命周期中，可能随时处于以下五种状态之一：

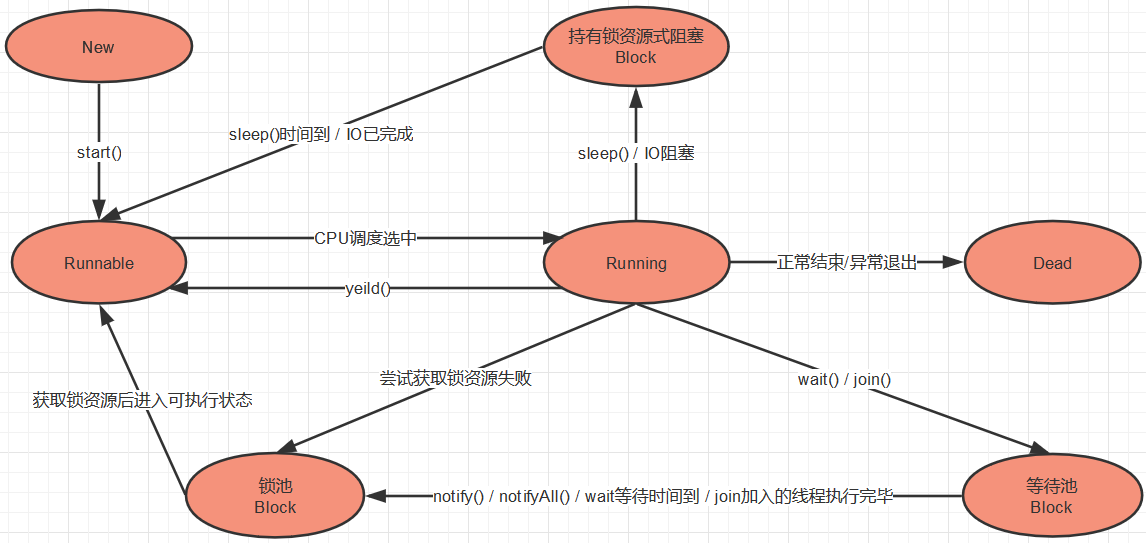
•创建（new）状态: 当用new操作符创建一个新的线程对象时，该线程处于创建状态，此时系统还没有为它分配资源。

•就绪（runnable）状态: 此时线程调用了start()方法，等待CPU进行调度

•运行（running）状态: 线程获得CPU资源，执行run()方法

•阻塞（blocked）状态: 因sleep()或wait()或其他线程join()或等待IO等原因，暂时停止执行，让出资源进入阻塞状态

•终止（dead）状态: 线程执行完毕，或被杀死，此时线程销毁



实现Runnable接口或继承Thread可以得到一个线程类，new一个该类的实例出来，线程就进入了**创建（new）状态**

调用线程的start()方法，此线程进入**就绪（runnable）状态**。就绪状态只是说该线程处于随时可运行的状态，但具体是否运行，何时运行由cpu决定，cpu没有”临幸”就只能等着。当时间片用完、sleep()结束、join()加入的其他线程执行完毕、IO输入完成、线程拿到别的线程释放的自己所需的同步锁等情况出现时，线程进入就绪状态。需要注意的是，yield()方法是线程主动**建议**cpu结束自己的时间片，让给其他线程执行，但是否采纳建议，由CPU决定

线程调度程序从可运行池中选择一个线程，为其分配cpu资源，进入**运行（running）状态**。这也是线程进入运行状态的唯一一种方式。

因某种原因，线程进入不可继续执行状态，这种状态被称为**阻塞（blocked）状态**

以下情况会出现线程阻塞状态：

1.线程调用sleep()方法，主动放弃占用的处理器资源，但不会释放自己所持有的锁资源，等待sleep()时间到重新进入可执行状态

2.线程等待IO输入，等待IO输入完成重新进入可执行状态，IO阻塞是不会释放锁的

3.线程试图获得一个同步锁(synchronized)，但该同步锁正被其他线程所持有，等待获取锁后重新进入可执行状态。

4.线程被wait()方法阻塞同时会释放自己持有的同步锁，然后等待notify()方法唤醒或wait()时间到，此时进入同步锁阻塞状态(第3种阻塞情况)，获取到锁之后才能进入可执行状态

5.线程被其他线程通过join()方法打断执行，待通过join()加入的线程执行完毕后再重新进入可执行状态，由于join()底层是wait()方法，因此join()会释放锁

当线程的run()方法完成时，或者线程抛出一个未捕获的Exception或Error，或者通过stop()方法结束该线程(stop极度危险)，我们就认为它进入**终止（dead）状态**。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦终止了，就不能复生。在一个终止的线程上再次调用start()方法，会抛java.lang.IllegalThreadStateException异常。

### 6.1.6线程的优先级

在多线程开发中，不同的线程对于程序的重要性不同，我们根据其重要性对线程进行基于权重的划分，这就是线程的优先级。Java的每一个线程都有自己的优先级，其范围是1~10，默认为5。高优先级的线程会优先于低优先级的线程执行，但这并不能保证线程的执行顺序，也就是说，高优先级相较于低优先级更容易获得cpu资源，但也有可能出现低优先级比高优先级线程先执行的情况，因此优先级机制并不能保证高优先级线程一定比低优先级先执行。

可以通过调用线程的setPriority()方法来设置线程优先级，设置线程优先级应当尽量放在线程开始前

### 6.1.7守护线程

Java的线程分为两种：User Thread(用户线程)、Daemon Thread(守护线程)。Daemon守护线程的作用是为其他线程提供便利服务，只要当前JVM实例中尚存任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。比如JVM的GC垃圾回收线程就是一条典型的守护线程

User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全部退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。因为没有了被守护者，Daemon也就没有工作可做了，也就没有继续运行程序的必要了。

可以通过 setDaemon()方法设置守护线程

不要尝试使用守护线程来完成资源回收的工作，因为所有的User Thread结束时，守护线程不一定会执行到回收资源的代码处

### 6.1.8死锁

线程的安全性和活跃性之间存在某种平衡，我们通过加锁保证线程安全，但加锁过多或加锁不当会导致线程活跃性故障，如：死锁

关于死锁有一个很经典的哲学家进餐问题：

五位哲学家去吃午餐，坐在一张圆桌旁，桌上有五根筷子，每两人之间摆一根，他们只能拿自己左右手边的筷子，每位哲学家要么思考要么进餐，当某位哲学家进餐时，他必须拿到两只筷子，每一位哲学家在未拿到筷子前，不会放弃手中的筷子，当哲学家进餐完毕后，他必须将手中的筷子放回原处

假设出现这么一种情况，每个哲学家都拿着左手的筷子，永远都在等右边的筷子(或者相反)，我们就称其为死锁

**死锁：两个或两个以上的进程在执行过程中，互相持有对方所需要的资源，而又相互等对方释放锁，而造成的一种阻塞的现象**

**资源独占这个条件一般不可破坏，一般破坏后三种**

**方法1：所有线程在运行之前，必须一次性地申请在整个运行过程中所需的全部资源，这可以破坏请求与保持条件，但资源利用效率太低**

**方法2：每个线程提出新的资源申请前，释放它所占有的资源，这可以破坏请求与保持条件，但是容易发生活锁**

**方法3：每个线程在获取新资源失败时，必须释放自己已持有的资源，这可以破坏不剥夺条件**

**方法4：修改加锁顺序，破坏循环等待条件**

### 6.1.9活锁、饥饿、无锁

除了死锁外，多线程使用不当还会产生如下问题：

**活锁：**和死锁相反，活锁是指两个或两个以上的进程在执行过程中都拿到了对方所需的资源，但又互相谦让，然后再次获得对方所需资源，再次谦让…线程始终无法正常执行的现象

死锁会导致发生死锁的线程处于阻塞状态，互相等待对方释放锁，而活锁虽然不会阻塞相关线程，却一直处于不断尝试加锁的状态，实际上是白白浪费CPU资源

**饥饿：**因为某种原因，一条线程始终无法无法获取CPU执行，这种原因可能是优先级高的线程一直抢占优先级低的线程资源导致的，也可能是某个线程始终占用其他线程资源不放导致的

**无锁**：即没有对资源进行锁定，即所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。无锁典型的特点就是一个修改操作在一个循环内进行，线程会不断的尝试修改共享资源，如果没有冲突就修改成功并退出否则就会继续下一次循环尝试(自旋)。如CAS就是一种无锁技术

* 1. 实现多线程的方法

实现多线程有三种方式，一种是实现Runnable接口并重写run()方法，另一种是继承Thread类并重写run()方法，最后一种是实现callable接口并重写call()方法，下面是这三种方式的对比：

### 6.2.1继承Thread类方式

优点：编写简单，如果需要访问当前线程，无需使用Thread.currentThread()方法，直接使用this，即可获得当前线程。

缺点：因为线程类已经继承了Thread类，所以不能再继承其他的父类。

Eg：

public class LiftOff extends Thread {

protected int countDown = 100;  
 private static int *taskCount* = 0;  
 private final int id = *taskCount*++;

public LiftOff() {}  
 public LiftOff(int countDown) {  
 this.countDown = countDown;  
 }  
 public String status() {  
 return "id:" + id + "(" + (countDown > 0 ? countDown : "发射!") + ")";  
 }

@Override  
 public void run() {  
 while(this.countDown-->0){  
 System.*out*.println(this.status());  
 }  
 }  
}

public class Test {

@org.junit.Test  
 public void test() {  
 LiftOff thread=new LiftOff();  
 thread.start();  
 }  
}

### 6.2.2实现Runnable接口的方式

优点：线程类只是实现了Runable接口，还可以继承其他的类。在这种方式下，可以多个线程共享同一个目标对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

缺点：编程稍微复杂，如果需要访问当前线程，必须使用Thread.currentThread()方法。

Eg：

public class LiftOff implements Runnable {

protected int countDown = 100;  
 private static int *taskCount* = 0;  
 private final int id = *taskCount*++;

public LiftOff() {}  
 public LiftOff(int countDown) {  
 this.countDown = countDown;  
 }  
 public String status() {  
 return "id:" + id + "(" + (countDown > 0 ? countDown : "发射!") + ")";  
 }

@Override  
 public void run() {  
 while(this.countDown-->0){  
 System.*out*.println(this.status());  
 }  
 }  
}

一般来说，run()方法通常是一个循环，线程将不断执行循环体直到出现某个条件跳出循环，这时run()方法执行完毕，线程结束

public class Test {

@org.junit.Test  
 public void test() {  
 LiftOff thread=new LiftOff();  
 new Thread(thread).start();  
 }  
}

### 6.2.3实现callable接口的方式

Callable接口实际上是属于Executor框架中的功能类，Callable接口与Runnable接口的功能类似，但提供了比Runnable更加强大的功能。

**public interface Callable<V> {**

**V call() throws Exception;  
}**

**public interface Runnable {**

**public abstract void run();  
}**

他们之间的不同：

1.实现Callable接口的任务线程能返回执行结果；而实现Runnable接口的任务线程不能返回结果；

2.Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛；

class Thread\_1 implements Callable<String> {

private int num = 1000;  
 @Override  
 public String call() throws Exception {  
 while (this.num-- > 0) {  
 System.*out*.println("num = " + num);  
 }  
 return String.*valueOf*(num);  
 }  
}

@Test

public void test\_1() {

Executor executor=Executors.*newSingleThreadExecutor*();  
 Callable<String> callable=new Thread\_1();  
 FutureTask<String> task = new FutureTask<>(callable);  
 executor.execute(task);  
 try {

// get()方法将会阻塞直到任务执行完毕，然后将结果返回  
 System.*out*.println("task.get() = " + task.get());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
}

* 1. 线程常用方法

### 6.3.1 yeild()

线程调用yield()方法意味着线程告诉JVM自己正在执行的任务并不紧急，”**暗示**”虚拟机自己可以让出CPU资源给其他线程使用，但具体是否会出让资源并不保证，由CPU决定是否切换线程。当然了，即使出让CPU，线程也是转入可执行状态，并不会释放自己所持有的锁资源，随时准备再次执行

### 6.3.2 wait()和notify()/notifyAll()

**这三个方法都是java.lang.Object的方法**

在讨论这三种方法之前，首先讨论两个概念，这两个概念实际上我们之前也有提过

**锁池:** 假设线程A已经拥有了某个锁资源，而其它的线程想要调用这个锁资源对象的某个synchronized方法(或者synchronized块)，由于这些线程在进入synchronized方法(或者synchronized块)之前必须先获得该锁的拥有权，但是该锁目前正被线程A拥有，所以这些线程就进入了该对象的锁池中，当锁资源被线程A释放后，锁池中的线程会竞争 锁资源，获取到锁的线程将进入可执行状态。参见6.1.5导致阻塞的第三种情况

**等待池:**假设一个线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁后，进入到了该对象的等待池中，等待池中的线程在被唤醒前不会竞争锁资源。参见6.1.5导致阻塞的第四种情况

对某个 锁资源对象 调用**wait()方法**会使当前线程进入该锁对象的等待池中，并释放线程自己所持有的该 锁资源对象 的锁，直到其他线程调用此 锁资源对象 的notify()方法或 notifyAll()方法唤醒它。wait()方法当然也可以设置一个long类型的等待时间，一旦超时也会自动唤醒

对 锁资源对象 调用**notify()方法**，将从该 锁资源对象 的等待池中随机选择一个线程进入该 锁资源对象 的锁池中，和锁池中的其他资源竞争锁

对 锁资源对象 调用**notifyAll ()方法**，会将该 锁资源对象 的等待池中所有的线程都唤醒到锁池中一起来竞争锁资源

需要注意的是，由于wait()方法会释放自己持有的锁资源，因此，该方法必须放在synchronized同步代码块中使用，否则抛出IllegalMonitorStateException异常，notify/notifyAll方法也是如此

另外，调用notify/notifyAll方法后，被唤醒的线程也不会立即进入可执行状态，只是被从锁对象ObjectMonitor的等待池中移入了锁对象的锁池，此时该线程会尝试和锁池中的其他线程竞争锁资源，如果获取锁成功，才能进入可执行状态

class Waiter implements Runnable{

private String msg;

public Waiter(String msg) {  
 super();  
 this.msg = msg;  
 }

@Override  
 public void run() {  
 synchronized (msg) {  
 System.*out*.println(Thread.*currentThread*().getName()+"获取msg控制权");  
 try {  
 Thread.*sleep*(10);  
 msg.wait();  
 System.*out*.println(Thread.*currentThread*().getName()+"被恢复");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

class Notifyer implements Runnable {  
 private String msg;

public Notifyer(String msg) {  
 super();  
 this.msg = msg;  
 }

@Override  
 public void run() {  
 synchronized (msg) {  
 System.*out*.println("notify开始工作");  
 try {  
 Thread.*sleep*(100);  
 msg.notify();  
 System.*out*.println("notify工作完成");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

@Test  
 public void test\_2() {  
 String msg=new String("------");  
 new Thread(new Waiter(msg),"waiter").start();  
 new Thread(new Notifyer(msg),"notify").start();  
 while (true){}  
 }

**notify()在使用不当时可能导致异常**，如下例：

//生产者

class Producer implements Runnable {  
 private String threadName;  
 //缓冲队列，本例中缓冲队列最多只会有一个产品待消费  
 List<Integer> cache;  
  
 public Producer(List<Integer> cache,String threadName) {  
 this.cache = cache;  
 this.threadName=threadName;  
 }  
  
 @Override  
 public void run() {  
 //生产者线程一旦被唤醒就会不断生产  
 while (true) {  
 produce();  
 }  
 }  
  
 private void produce() {  
 synchronized (cache) {  
 try {  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*生产者"+this.threadName+"获取锁\*\*\*\*\*\*\*\*\*\*" );  
 //首先判断缓冲队列中是否已存在数据，如果存在就把自己移入缓冲队列的等待池中  
 while (cache.size() == 1) {  
 System.*out*.println("缓存不为空，生产者"+this.threadName+"进入等待池" );  
 cache.wait();  
 }  
 // 模拟一秒生产一条消息  
 Thread.*sleep*(1000);  
 System.*out*.println("生产者"+this.threadName+"生产数据！");  
 cache.add(new Random().nextInt());  
 //从等待池中随机唤醒一个线程  
 cache.notify();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*生产者 "+this.threadName+"从等待池中随机唤醒一个线程并释放锁\*\*\*\*\*\*\*\*\*\*");  
 }  
 catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}  
class Consumer implements Runnable {  
 private String threadName;  
 //缓冲队列，本例中缓冲队列最多只会有一个产品待消费  
 List<Integer> cache;  
  
 public Consumer(List<Integer> cache,String threadName) {  
 this.cache = cache;  
 this.threadName=threadName;  
 }  
  
 @Override  
 public void run() {  
 //消费者线程一旦被唤醒就会不断消费  
 while (true) {  
 consume();  
 }  
 }  
  
 private void consume() {  
 synchronized (cache) {  
 try {  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*消费者"+this.threadName+"获取锁\*\*\*\*\*\*\*\*\*\*" );  
 //首先判断缓冲队列中是否为空，如果没有产品就把自己移入缓冲队列的等待池中  
 while (cache.isEmpty()) {  
 System.*out*.println("缓存为空，消费者"+this.threadName+"进入等待池" );  
 cache.wait();  
 }  
 //消费产品  
 System.*out*.println("消费者 "+this.threadName+"消费了数据"+cache.remove(0) + "！");  
 cache.notify();  
 System.*out*.println("\*\*\*\*\*\*\*\*\*\*消费者 "+this.threadName+"从等待池中随机唤醒一个线程并释放锁\*\*\*\*\*\*\*\*\*\*");  
 }  
 catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

如上例的消费者和生产者，如果我们只启动一个消费者和一个生产者两条线程，那么一切正常，等待池中始终只有一个线程，要么就是生产者唤醒消费者，要么消费者唤醒生产者，不会发生异常：

@Test

public void test() {

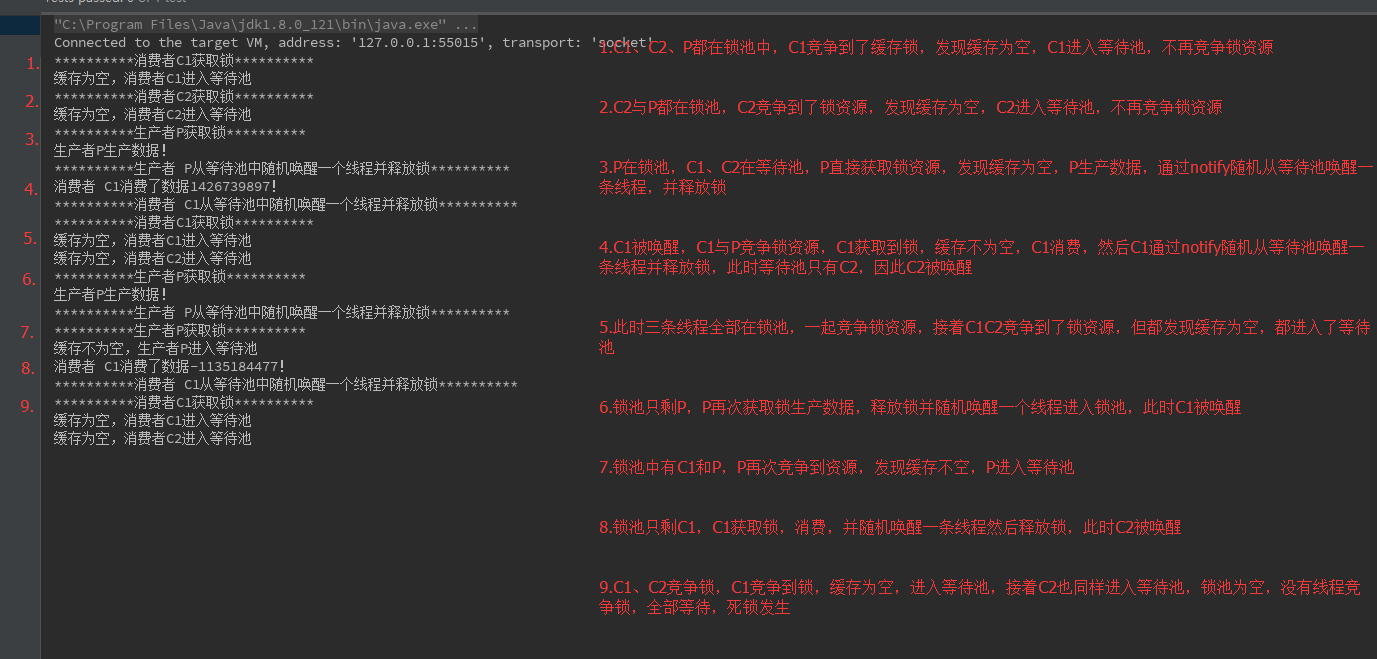
//一个消费者一个生产者  
 List<Integer> cache=new ArrayList<>();  
 new Thread(new Consumer(cache,"C")).start();  
 new Thread(new Producer(cache,"P")).start();  
 while (true){}  
}

如果开启了一个生产者和两个消费者，且使用notify()的方式来唤醒线程，那么则可能发生异常

@Test

public void test\_2() {  
 List<Integer> cache=new ArrayList<>();  
 new Thread(new Consumer(cache,"C1")).start();  
 new Thread(new Consumer(cache,"C2")).start();  
 new Thread(new Producer(cache,"P")).start();  
 while (true){}  
}

输出结果如下：



解决方法是将notify换成notifyAll

### 6.3.3 interrupt()

一个线程不应该由其他线程来强制中断或停止，而是应该由线程自己自行停止。interrupt翻译过来是”终断，打断”的意思，它被用于终止线程，但 interrupt()并不直接中断线程，而是通过给线程设置一个中断标识(每个线程中都有一个与中断相关联的 boolean 属性，用来表示线程的中断状态)，传达”线程应该被中断了”的信息，然后由我们自己编写的代码进行中断检查，再决定是否中断，他只是设置了一个特殊的标识位，并未起到任何实际作用。(实际上，interrupt()不能不能真正意义上地中断正在运行过程中的线程，但能通过抛出中断异常的方式真正中断处于阻塞状态中的线程)

那么哪些场景中需要用到中断机制呢？

1.很多线程的运行模式是死循环，比如在生产者/消费者模式中，消费者主体就是一个死循环，它不停的从队列中接受任务，执行任务，在停止消费者程序时，我们需要一种”优雅”的方法以关闭该线程。

2.抢12306的火车票，我们可能开启多个线程购买火车票，只要有一个线程买到了，我们就需要中断其他线程。

3.从第三方服务器查询一个结果，我们希望在限定的时间内得到结果，如果得不到，我们会希望取消该任务。此时需要中断查询的线程

……

在了解中断机制前，我们先来看看与中断标识相关的三个方法：

public boolean **isInterrupted**() //返回线程当前的中断状态  
public void **interrupt**() //设置线程的中断状态(设为true)  
public static boolean **interrupted**()//返回线程当前的中断状态，然后清除中断状态(设为false)

对于处于不同状态的线程，调用 interrupt()的效果是不一样的：

1. 对处于运行(running)状态的线程使用 interrupt()

interrupt()只是会设置线程的中断标志位，没有任何其它作用，不会打断线程的执行，为了能够正常终止线程，编写线程时应该在合适的位置检查中断标志位，比如：

//生产者

class Producer implements Runnable {  
 private String threadName;  
 //缓冲队列，本例中缓冲队列最多只会有一个产品待消费  
 List<Integer> cache;  
  
 public Producer(List<Integer> cache,String threadName) {  
 this.cache = cache;  
 this.threadName=threadName;  
 }  
  
 @Override  
 public void run() {  
 long beginTime=System.*currentTimeMillis*();  
 //生产者线程会不断生产,直至被打断  
 while (!Thread.*currentThread*().isInterrupted()) {  
 //5s后打断生产者  
 if(System.*currentTimeMillis*()-beginTime>500){  
 Thread.*currentThread*().interrupt();  
 System.*out*.println("打断线程"+Thread.*currentThread*().isInterrupted());  
 }  
 produce();  
 }  
 }  
  
 private void produce() {  
 synchronized (cache) {  
 try {  
 cache.add(new Random().nextInt());  
 System.*out*.println("生产者生产了数据" );  
 }  
 catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

@Test

public void test() {  
 List<Integer> cache=new ArrayList<>();  
 Thread thread=new Thread(new Producer(cache,"P"));  
 thread.start();  
 while (true){}  
}

1. 对处于阻塞(block)状态的线程使用 interrupt()

对于调用了sleep(), wait(), join()等方法而进入阻塞状态的线程，若此时调用线程的interrupt()将线程的中断标记设为true。由于处于阻塞状态，中断标记会被清除(重置为false)，同时抛出一个InterruptedException异常，从而退出阻塞(block)状态(抛出异常会导致被阻塞的线程重新开始执行)

@Override

public void run() {  
 try {  
 while (true) {

// 执行任务...  
 }  
 } catch (InterruptedException ie) {  
 // 由于产生InterruptedException异常，退出while(true)循环，线程终止！  
 }  
}

Thread.stop, Thread.suspend, Thread.resume等中断线程的方法极度危险，现行版本的jdk已将其标为废弃方法，应尽量避免使用

### 6.3.4 sleep()

Thread.sleep(long millis)和Thread.sleep(long millis, int nanos)静态方法强制当前正在执行的线程休眠（暂停执行），以“减慢”线程执行速度。

和wait()方法不同，线程睡眠并不会释放锁资源。因此当线程睡眠时间到期后，线程不需要进入锁池获取锁，而是直接进入可执行状态等待cpu调度

需要注意的是**sleep()中指定的时间是线程暂停运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就立即开始执行。**

### 6.3.5 join()

在某些情况下，主线程创建并启动了子线程，如果子线程中需要进行大量的耗时运算，主线程往往将早于子线程结束之前结束，如果主线程想等待子线程执行完毕后，获得子线程中的处理完的某个数据，就要用到join方法了，它等待调用join方法的子线程结束，再继续执行主线程，这使得主线程和子线程的并行执行变为串行执行

**join底层是wait方法，所以子线程调用该方法时，暂停主线程的同时也会释放主线程持有对象锁的**，而sleep在同步的方法中是不释放对象锁的，只有同步方法执行完毕，其他线程才可以执行。

### 6.3.6 currentThread ()

Thread类提供了一个静态方法currentThread ()用以获得当前正在执行的线程，有点类似this

### 6.3.7 isAlive()

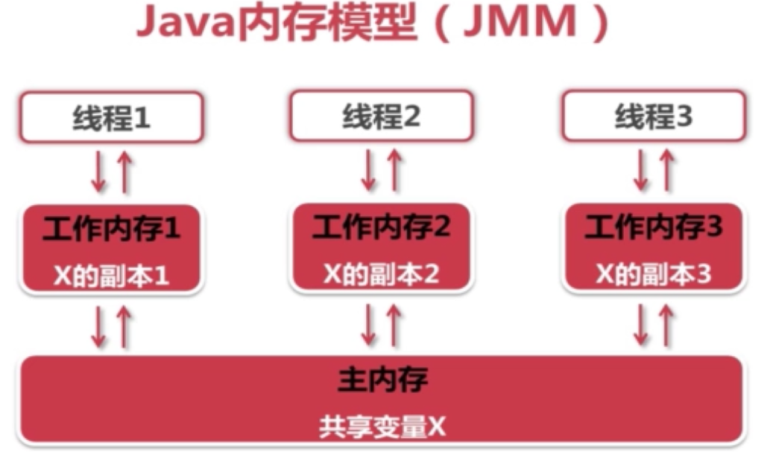
线程处于创建(new)状态时，调用isAlive()方法返回false。在线程的run()方法结束之前，即没有进入死亡状态之前，线程调用isAlive()方法返回true.

* 1. 并发的三个特性
     1. 可见性

**共享变量：如果一个变量在多个线程的工作内存中都存在副本，那么这个变量就是这几个线程的共享变量。**

**共享变量可见性：如果线程对共享变量的修改能够被其他线程及时看到，就可以实现共享变量在线程之间的可见性。**

JVM中存在一个主存区(Main Memory或Java Heap Memory)，对于所有线程进行变量共享，但线程不能直接操作主内存中的变量，每个线程都有自己独立的工作内存(Working Memory)，里面保存该线程使用到的变量的副本(主内存中该变量的一份拷贝)，**线程对所有变量的操作并非发生在主存区，而是发生在工作内存中，线程之间的工作内存是不能直接相互访问的，变量在程序中的传递，是依赖主存来完成的**。



上图是java内存模型JMM

基于这个原理线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1、线程A把工作内存A中更新的共享变量刷新到主内存中去。

2、线程B到主内存中去读取线程A之前已更新过的共享变量。

public class Test {

//一般情况下，多线程不能保证共享变量可见性  
 private static boolean *run*=true;  
  
 @org.junit.Test  
 public void test() {  
 Thread thread\_1=new Thread(() -> {  
 System.*out*.println("Ordinary A is running...");  
 while (*run*) ;  
 System.*out*.println("Ordinary A is terminated.");  
 });  
 Thread thread\_2=new Thread(() -> {  
 System.*out*.println("Ordinary B is running...");  
 *run*=false;  
 System.*out*.println("Ordinary B is terminated.");  
 });  
 thread\_1.start();  
 try {  
 Thread.*sleep*(10); //暂停主线程执行，让线程1继续执行10ms  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 thread\_2.start();  
 while(true){}  
 }  
}

运行结果：

Ordinary A is running...

Ordinary B is running...

Ordinary B is terminated.

从结果观察，发现线程B运行结束了，也就是说已经修改了共享变量stop的值。但是线程A还在运行，也就是说线程A并没有用接收到stop=true这个修改。

**通过volatile关键字保证可见性。**

**通过synchronized关键字定义同步代码块或者同步方法保障可见性。**

**通过Lock接口保障可见性。**

**通过Atomic原子类型保障可见性。**

* + 1. 有序性

**JVM存在指令重排，所以存在有序性问题。**

**指令重排序：**代码书写的顺序与实际执行的顺序不同，指令重排序是编译器或CPU为了提高程序性能而做的优化。这种重排序在单线程执行时并不会带来可见性问题，但放在多线程中，一切就不一样了。

比如：flag变量是个标记，用来标识变量a是否已被写入

class ReorderExample implements Runnable {

static int a = 0;  
 static boolean flag = false;  
  
 public void writer() {  
 a = 1; // 操作1  
 flag = true; //操作2  
 }  
  
 public void reader() {  
 if (flag) { //操作3  
 int i = a \* a; //操作4  
 }  
 }

public void run() {

//……………

}  
}

假设a是共享变量，有两个线程A和B，A首先执行writer()方法，随后B线程接着执行reader()方法。线程B在执行操作4时，能否看到线程A在操作1对共享变量a的写入？答案是：不一定能看到。由于操作1和操作2没有数据依赖关系，编译器和处理器可以对这两个操作重排序；同样，操作3和操作4没有数据依赖关系，编译器和处理器也可以对这两个操作重排序

假设A、B线程对上例的执行顺序是：2 -> 3 -> 4 -> 1(对操作1和操作2进行了重排序)，程序执行时，线程A首先写标记变量flag，随后线程B读这个变量。由于条件判断为真，线程B将读取变量a。此时，变量a还根本没有被线程A写入，在这里多线程程序的语义被重排序破坏了！

**通过synchronized关键字定义同步代码块或者同步方法保障有序性。**

**通过Lock接口保障有序性。**

**volatile关键字可在一定程度上保证有序性。（这里只能保证volatile所修饰的变量之前的程序不会在该变量之后执行，该变量之后的代码不会在变量之前执行。）**

#### 6.4.2.1 happends-before原则

为了保证在不改变java程序的语义下，尽可能减少对编译器和处理器重排序的限制，JMM在jdk5时引入了happends-before原则，重排序必须遵守happends-before原则 ：

1. **程序顺序规则：在同一个线程中，按照程序顺序，前面的操作Happends-Before于后续的任意操作，这保证了单线程内语句的有序性**
2. **Volatile规则:对于一个volatile对象的写操作，happends-before于程序后面对这个变量的读操作**
3. **传递规则：如果操作A happends-before 操作B，操作B happends-before 操作C，则操作A happends-before 操作C**
4. **锁规则：对于一个synchronized锁的解锁，happends-before于随后对这个锁的加锁操作。**这意味着如果线程1解锁了monitor a，接着线程2锁定了a，那么，线程1解锁a之前的写操作都对线程2可见（线程1和线程2可以是同一个线程），eg

class Test {  
 private int x = 0;  
 public void initx() {  
 synchronized (this) {  
 if (this.x < 10) {  
 x = 10;  
 }  
 }  
 }  
}

**如上所示，线程A将x修改为10后，解锁，然后线程B才可以对this加锁**

1. **线程启动规则：Thread对象的start()方法happends-before于此线程的每一个操作。**这意味着如果在线程A中启动了线程B，则线程B能够看到线程A在启动它之前的所有操作
2. **线程中断规则:对于线程interrupt()方法的调用happends-before于被中断线程的代码检测到中断事件的发生。**这意味着线程外调用的interrupt()方法，如果在时间线上先于IsInterrupt()执行，则计算机不会将其重排序到IsInterrupt()之后执行，eg：

class ThreadMain{  
 private int x = 0;  
 public void execute(){  
 Thread childThread = new Thread(()->{  
 if(Thread.*currentThread*().isInterrupted()){  
 //输出x = 100  
 System.*out*.println("x = " + x);  
 }  
 });  
 childThread.start();  
 x = 100;  
 childThread.interrupt();  
 }  
}

1. **线程终止规则：如果ThreadA调用了ThreadB.join()方法，那么ThreadB的所有操作，都happends-before线程A在调用ThreadB.join()后的操作**
2. **对象终结规则:一个对象的初始化完成，happends-before于它的finalize()方法开始之前**
   * 1. 原子性

原子性：即一个操作或者多个操作 要么全部执行，要么就都不执行，并且执行的过程不会被任何因素打断。

在Java中，对基本数据类型的变量的读取和赋值操作都是原子性操作。**(除了64位的long和double)**

下列操作中，只有操作1和操作2是原子性的

a = true; //操作1

a = 5; //操作2

a = b; //操作3，包含两个操作，1.读取b的值；2.将b的值赋值给a。

a = b + 2; //操作4，包含三个操作，1.读取b的值；2.计算b+2；3.将b+2的计算结果赋值给a。

a ++; //操作5，包含三个操作，读取a的值；2计算a+1；3.将a+1的计算结果赋值给a。

**通过synchronized关键字定义同步代码块或者同步方法保障原子性。**

**通过Lock接口保障原子性。**

**通过Atomic类型保障原子性。**

**Java内存模型要求：基本数据类型的变量的读取和写入必须是原子性的，但对于没有被volatile声明过的64位long和double存在例外，这是因为JVM允许将64位变量的读操作和写操作分为两个32位的操作执行。这可能会导致多线程情况下，读取/写入某个没有被volatile声明的long或double时，读到/写入某个变量的高32位和另一个变量的低32位。volatile本身不保证获取和设置操作的原子性，仅仅保持修改的可见性。但是java的内存模型保证声明为volatile的long和double变量的读写操作是原子的。不过，这种情况非常罕见，目前各平台的商用虚拟机几乎都选择把64位的读写作为原子操作来实现规范的。(参考《深入理解Java虚拟机》)**

* + 1. volatile
       1. volatile的作用

Java为我们提供了volatile关键字来保证共享变量的可见性，**volatile只能修饰类的成员变量，包括静态和非静态**

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了以下语义：

**1.保证了共享变量的可见性，它强制线程每次对共享变量执行写操作后，都要立即将其修改值同步到主存中；并且写操作会导致共享变量在其他线程中的缓存失效，其他线程使用缓存时，发现本地工作内存中共享变量无效，就需要从主内存中获取**

**2. 通过插入内存屏障，一定程度上保证了有序性，详见happends-before原则**

**3.** **保证double和long在32位操作系统中的读写原子性**

public class Test {

//一般情况下，多线程不能保证共享变量可见性  
 private static boolean *run*=true;  
  
 @org.junit.Test  
 public void test() {  
 Thread thread\_1=new Thread(() -> {  
 System.*out*.println("Ordinary A is running...");  
 while (*run*) ;  
 System.*out*.println("Ordinary A is terminated.");  
 });  
 Thread thread\_2=new Thread(() -> {  
 System.*out*.println("Ordinary B is running...");  
 *run*=false;  
 System.*out*.println("Ordinary B is terminated.");  
 });  
 thread\_1.start();  
 try {  
 Thread.*sleep*(10); //暂停主线程执行，让线程1继续执行10ms  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 thread\_2.start();  
 while(true){}  
 }  
}

在我们试图对线程采取中断操作时，我们往往会通过类似上例中设置中断标记的方式来进行中断，但这段代码并不一定会将线程中断，也许在大多数时候，这个代码能够把线程中断(本例中为了方便重现线程无法中断的情况，进行了特殊处理)，但是也有可能会导致无法中断线程（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

但是在用volatile修饰之后就变得不一样了：

第一：使用volatile关键字会强制将修改的值立即写入主存；

第二：使用volatile关键字的话，当线程2进行修改时，会将run值写入主存，使得线程1的工作内存中缓存变量run的缓存无效，强制其到主存中读取run

然而volatile并不能保证原子性

class Test2 {

public volatile int inc = 0;  
  
 public void increase() {  
 inc++;  
 }  
  
 public static void main(String[] args) {  
 final Test2 test = new Test2();  
 for(int i=0;i<10;i++){  
 new Thread(() -> {  
 for(int j=0;j<1000;j++)  
 test.increase();  
 }).start();  
 }  
  
 while(Thread.*activeCount*()>1) { //保证前面的线程都执行完  
 Thread.*yield*();  
 }  
 System.*out*.println(test.inc);  
 }  
}

假如某个时刻变量inc的值为10，线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1让出cpu；然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，所以线程2会直接去主存读取inc的值，发现inc的值是10，然后进行加1操作，并把11写入工作内存，最后写入主存。然后线程1接着进行加1操作，由于已经读取了inc的值，不需要再去读取，直接加就可以了，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。由此看来**volatile并不能保证原子性**

如果将上例中的int换成AtomicInteger是可以正常执行的，因为AtomicInteger的自增是原子性操作

**一定要注意的是：对于volatile修饰的引用类型（包括对象、数组等非基本数据类型），其仅仅是保证引用地址的可见性，而不是引用指向的对象元素的可见性。**这是因为java对象模型不会将主内存中的对象整个复制到工作内存中的(假如有10MB大小也复制？)，仅仅是复制其引用地址，所以volatile只能保存其地址的可见性

* + - 1. happends-before原则

为了简化对重排序底层的理解，java提出了Happends-before原则

1. **程序顺序规则：在同一个线程中，按照程序顺序，前面的操作Happends-Before于后续的任意操作，这保证了单线程内语句的有序性**
2. **Volatile规则:对于一个volatile对象的写操作，happends-before于程序后面对这个变量的读操作**
3. **传递规则：如果操作A happends-before 操作B，操作B happends-before 操作C，则操作A happends-before 操作C**
4. **锁规则：对于一个synchronized锁的解锁，happends-before于随后对这个锁的加锁操作。**
5. **线程启动规则：Thread对象的start()方法happends-before于此线程的每一个操作。**这意味着如果在线程A中启动了线程B，则线程B能够看到线程A在启动它之前的所有操作
6. **线程中断规则:对于线程interrupt()方法的调用happends-before于被中断线程的代码检测到中断事件的发生。**这意味着在线程外调用的interrupt()方法必然在线程内部的isInterrupt()方法之前执行
7. **线程终止规则：如果ThreadA调用了ThreadB.join()方法，那么ThreadB的所有操作，都happends-before线程A在调用ThreadB.join()后的操作**
8. **对象终结规则:一个对象的初始化完成，happends-before于它的finalize()方法开始之前**
   * + 1. cas+volatile实现同步锁

CAS只能同步一个变量的修改，我们又应该如何用它来锁住代码块呢？

同样的，volatile并不能保证共享变量操作的原子性，如何能够作为锁呢？

先说说实现锁的要素

1. **同步代码块在同一时刻内只能有一个线程能执行**
2. **加锁操作要happens-before同步代码块里的操作，而代码块里的操作要happens-before解锁操作**
3. **同步代码块结束后相对其他线程其修改的变量是可见的 (内存可见性)**

要素1：可以利用CAS的原子性来实现，任意时刻只有一个线程能成功操作变量 先设想CAS操作的共享变量是一个关联代码块的同步状态变量，同步开始之前先CAS更新状态变量(共享变量)为加锁状态，同步结束之后，再CAS状态变量为无锁状态。如果期间有第二个线程来加锁，则会发现状态变量为加锁状态，则放弃执行同步代码块

要素2：使用volatile修饰状态变量，禁止指令重排 volatile保证同步代码里的操作happens-before解锁操作，而加锁操作happens-before代码块里的操作

要素3：还是用volatile，volatile变量写指令前后会插入内存屏障， volatile修饰的状态变量被CAS为无锁状态前，同步代码块的脏数据就会被更新，被各个线程可见

伪代码：

volatile state = 0 ; // 0-无锁 1-加锁；volatile禁止指令重排，加入内存屏障  
...  
if(cas(state, 0 , 1)){ // 1 加锁成功，只有一个线程能成功加锁  
 ... // 2 同步代码块  
 cas(state, 1, 0); // 3 解锁时2的操作具有可见性  
}

**最典型的例子就是AQS框架了**

* + - 1. volatile的底层实现

**我们知道，volatile有两个功能：**

1. **保证可见性**
2. **禁止指令重排序**
   * + - 1. volatile可见性底层实现（MESI协议）

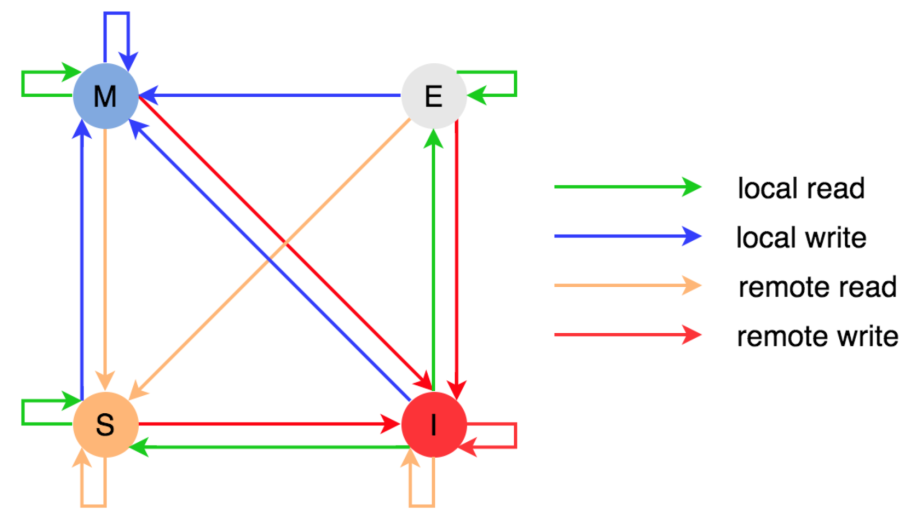
缓存一致性协议有很多种，这里仅介绍最常用的Intel的MESI协议

CPU操作缓存的单位是”缓存行”(cacheline)，如果CPU希望读取一个缓存的变量x，那么CPU需要读取变量x所在的整个缓存行cacheline，一般一个缓存行是64字节

**缓存行cacheline的元信息中有一个flag字段，它对应4个状态：**

|  |  |  |
| --- | --- | --- |
|  | **cacheline的中的数据和主内存中不一致** | **cacheline中的数据和主内存中的数据内容一致** |
| **cacheline中的数据仅被当前cpu核心缓存** | M（Modified） | E（Exclusive） |
| **除了当前cpu核心外，还有其他cpu核心缓存了该cacheline** | I（Invalid）代表缓存行无效了 | S（Shared） |

**cacheline的这4个状态将根据cpu核心对cacheline的读写操作而进行不断转换，如下图：**



CPU和主内存通过总线（BUS）互通消息。

CPU感知其他CPU的行为（比如读、写某个缓存行）就是通过嗅探（Snoop）总线（BUS）中其他CPU发出的请求消息完成的，有时CPU也需要针对总线（BUS）中的某些请求消息进行响应。这被称为”**总线嗅探机制**”。

**CPU读请求**：缓存处于 M、E、S 状态都可以被读取，I 状态CPU 只能从主存中读取数据

**CPU写请求**：缓存处于 M、E 状态才可以被写。对于S状态的写，需要将其他CPU中缓存行置为无效才行。

每个cpu核心通过嗅探在总线上传播的数据来检查自己的缓存值是不是过期了，如果cpu核心发现自己的cacheline中的数据被其他cpu核心修改了，就会将当前处理器的缓存行设置无效**(Invalid)**状态，cpu核心对这个数据进行修改操作的时候，会重新从主内存中把数据读到处理器缓存中；同时volatile**强制线程每次对共享变量执行写操作后，都要立即将其修改值同步到主存中**，这样就实现了volatile的可见性

因此volatile是通过**总线嗅探机制**完成可见性保证的

**但由于总线嗅探机制会不断的监听总线，如果大量使用volatile 会不断地占用总线带宽，导致总线流量激增，引起总线风暴。所以，volatile 的使用要适合具体场景**

对于多个线程之间共享的多个热点数据，应避免这些热点数据在同一个缓存行里，解决思路有两个：

* + - 1. 填充字节：一个缓存行有64字节，可以存8个long，我们可以在中间再加几个long，保证把另一个热点数据挤出同一个缓存行
      2. @Contended注解：java8以后提供@Contended注解，同时需要虚拟机加上参数XX:-RestrictContended参数生效，原理是通过增加padding来确保当前变量独享缓存行

无论哪种方法都很浪费高速缓存的缓存空间，所以要谨慎使用

* + - * 1. 指令重排序

编译器或cpu为了优化程序执行性能，会对指令进行重排序，主要有两个阶段，共三种重排序

阶段1：编译器重排序

**编译器重排序**就是在编译过程中，编译器根据上下文分析对指令进行重排序，目的是减少CPU和内存的交互，重排序之后尽可能保证CPU从寄存器或缓存行中读取数据。

阶段2(part 1)：指令级并行重排序

在cpu内核中一般会有多个执行单元，比如算术逻辑单元、位移单元等。在引入并行指令集之前，CPU在每个时钟周期内只能执行单条指令，也就是说只有一个执行单元在工作，其他执行单元处于空闲状态；在引入并行指令集之后，CPU在一个时钟周期内可以同时分配多条指令在不同的执行单元中执行。

这种优化的本质是通过提前执行其他可执行指令来填补CPU的时间空隙，充分提高cpu执行单元的利用率，但是这种优化的前提是必须要保证在单线程情况下程序执行结果和预期一致。

阶段2(part 2)：内存系统重排序

MESI协议虽然可以实现缓存的一致性，但是也会存在一些问题：如果CPU\_0要对一个在缓存行中共享的变量x (S状态)进行写入，首先需要发送一个使该缓存行失效的指令给到其他缓存了该数据的 CPU，并且要等到他们的确认回执，CPU0在这段时间内都会处于阻塞状态。为了避免阻塞带来的资源浪费，CPU中又引入了store bufferes

引入store bufferes后，CPU\_0 只需要在写入共享数据x时，直接把x写入到 store bufferes中，同时向总线发送invalidate消息，然后继续去处理其他指令（异步） 。当cpu\_0收到其他所有 CPU 发送了invalidate ack消息时，再将store bufferes中的数据存储至缓存行中，最后再从缓存行同步到主内存。

但是这种优化就会带来了有序性问题，我们来看下面的例子

value = 10;//S->I状态，将value写入store bufferes，通知其他CPU当前value的缓存失效  
 isFinish = true;//E状态

假设value是共享变量且未被修改，即状态为S；isFinish是非共享变量且未被修改，即状态为E。如果cpu\_0执行上述代码，value被修改，从状态S->状态I，cpu\_0需要通知其他cpu变量value已经失效，并等待其他cpu的ack。在发送消息之前，cpu\_0会将数据写入store bufferes，然后立即执行**isFinish = true**语句，E状态的修改不需要通知其他cpu，因此很快，很可能value失效的ack完成之前，第二条语句就已经执行完毕了，这就造成了指令重排

* + - * 1. volatile禁止指令重排底层实现（内存屏障）

首先我们来看看，如果不禁止指令重排序，会出现哪些问题，我们将读和写操作进行排列组合，可以得到如下四种读写顺序问题：

|  |  |
| --- | --- |
| **名称** | **简介** |
| 读—读(LoadLoad) | 先执行load1，再执行load2。发生重排序后可能先执行load2再执行load1 |
| 写—写(StoreStore) | 先执行store1，再执行store2。发生重排序后可能先执行store2再执行store1 |
| 读—写(LoadStore) | 先执行load1，再执行store2。发生重排序后可能先执行store2再执行load1 |
| 写—读(StoreLoad) | 先执行store1，再执行load2。发生重排序后可能先执行load2再执行store1 |

如果不考虑重排序，读写的顺序不会乱，但是为了提高cpu效率，cpu会对语句执行顺序放松要求，不同的cpu对上表中的读写顺序放松程度不同，比如TSO模型，放松了读—写(LoadStore)一致性要求，而PSO模型放松了写—读(StoreLoad)一致性要求和写—写(StoreStore)一致性要求

Java为了屏蔽不同cpu对于读写顺序一致性要求的差异，抽象出以下4种内存屏障：

|  |  |  |
| --- | --- | --- |
| **内存屏障** | **指令示例** | **简介** |
| **LoadLoad屏障** | Load1;  LoadLoad屏障;  Load2; | 保证在Load2读取之前，Load1的数据加载完成 |
| **StoreStore屏障** | Store1;  StoreStore屏障;  Store2; | 保证在store2写入之前，store1的数据刷新到主存 |
| **LoadStore屏障** | Load1;  LoadStore屏障;  Store2; | 保证在store2写入的数据被刷新到主存之前，Load1的数据被读取完毕 |
| **StoreLoad屏障** | Store1;  StoreLoad屏障;  Load2; | 保证在Load2读取数据之前，store1写入的数据被刷新到主存 |

JMM通过在指令序列中插入以上四种内存屏障来实现volatile的有序性保证，

* + 1. synchronized
       1. synchronized如何保证原子性、可见性、有序性

Java内置了Synchronized关键字来提供线程同步机制

1. synchronized是同步锁，同步块内的代码相当于同一时刻单线程执行，故不存在原子性和指令重排序的问题
2. synchronized关键字的语义JMM有两个规定，保证其实现内存可见性：线程加锁前，将清空工作内存中共享变量的值，从主内存中读取最新值；线程解锁前，必须把共享变量的最新值刷新到主内存中；

因此：

**Synchronized可以保证线程的可见性、有序性和原子性**，volatile拥有的功能，synchronized都可以做到

Synchronized关键字有四种用法：

1. 修饰方法
2. 修饰静态方法
3. 修饰代码块
4. 修饰类
   * + 1. Synchronized使用
          1. 修饰方法

Synchronized修饰一个方法很简单，就是在方法的返回值前面加synchronized，如果多个线程试图调用该方法，只有一个能成功调用，其它线程就会被锁住，eg：

class MyThread implements Runnable {

public synchronized void test() {  
 System.*out*.println("test开始..");  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("test结束..");  
 }  
 public void get(){  
 System.*out*.println("访问get方法");  
 }  
  
 public void run() {  
 this.get();  
 this.test();  
 }  
  
 public static void main(String[] args) {  
 Runnable runnable = new MyThread();  
 for (int i = 0; i < 3; i++) {  
 new Thread(runnable).start();  
 }  
 }  
}

输出：

访问get方法

test开始..

访问get方法

访问get方法

test结束..

test开始..

test结束..

test开始..

test结束..

注意：

**1.虽然可以使用synchronized来定义方法，但synchronized并不属于方法定义的一部分，因此，synchronized关键字不能被继承。**如果在父类中的某个方法使用了synchronized关键字，而在子类中覆盖了这个方法，在子类中的这个方法默认情况下并不是同步的，必须给子类方法也加上synchronized才行

**2. synchronized修饰非静态方法锁住的是本对象(this)**

**3. 用它修饰普通方法，锁住的是this对象。因此当一个线程访问对象的一个synchronized方法时，另一个线程仍然可以访问该对象中的非synchronized方法，但不可以访问该对象的任何synchronized方法**

**用它修饰静态方法，锁住的是类对象。因此当一个线程访问类的一个静态synchronized方法时，另一个线程仍然可以访问该对象中的非staitc 的synchronized方法，但不可以访问该对象的任何static 的synchronized方法**

* + - * 1. 修饰代码块

Synchronized修饰代码块需要指定需要锁住的实例对象，eg:

class MyThread extends Thread {

public void test() {  
 synchronized(this){  
 System.*out*.println("test开始..");  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("test结束..");  
 }  
 }  
 public void run() {  
 this.test();  
 }  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 3; i++) {  
 Thread thread = new MyThread();  
 thread.start();  
 }  
 }  
}

上面的例子看似能够保证3个线程串行执行test方法，但实际上并没有，输出结果为：

test开始..

test开始..

test开始..

test结束..

test结束..

test结束..

这是因为synchronized修饰代码块时，锁住的是this(本线程)对象，3个线程锁的是不同的三个对象

注意：

**1. synchronized锁住的是括号里的对象，而不是代码。对于非static的synchronized方法，锁的就是对象本身也就是this。(同上*修饰方法*)**

**2.** **当一个线程访问对象的一个synchronized同步代码块时，另一个线程仍然可以访问该对象中的非synchronized同步代码块**

**3.也可以给某个对象加锁**

**Eg：**

class MyThread implements Runnable {

private Date date = new Date();  
  
 public void test() {  
 synchronized (date) {  
 System.*out*.println("test开始..");  
 try {  
 Thread.*sleep*(1000);  
 System.*out*.println("num = " + date);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("test结束..");  
 }  
  
 public void get() {  
 System.*out*.println("访问get方法");  
 }  
  
 public void run() {  
 this.get();  
 this.test();  
 }  
  
 public static void main(String[] args) {  
 Runnable runnable = new MyThread();  
 for (int i = 0; i < 3; i++) {  
 new Thread(runnable).start();  
 }  
 }  
}

输出结果：

访问get方法

test开始..

访问get方法

访问get方法

num = Mon Jul 23 14:36:40 CST 2018

test结束..

test开始..

num = Mon Jul 23 14:36:40 CST 2018

test结束..

test开始..

num = Mon Jul 23 14:36:40 CST 2018

test结束..

* + - * 1. 修饰静态方法或对象

由于静态方法是属于类的而不属于对象的，**synchronized修饰的静态方法锁定的是这个类实例化出来的所有对象**，这点与上面两种使用方式不同。

class MyThread implements Runnable {

private static Date *date* = new Date();  
  
 public void test() {  
 synchronized (*date*) {  
 System.*out*.println("test开始..");  
 try {  
 Thread.*sleep*(1000);  
 System.*out*.println("num = " + *date*);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("test结束..");  
 }  
  
 public void get() {  
 System.*out*.println("访问get方法");  
 }  
  
 public void run() {  
 this.get();  
 this.test();  
 }  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 3; i++) {  
 Runnable runnable = new MyThread();  
 new Thread(runnable).start();  
 }  
 }  
}

可以看到上面即使new了三次线程，依然能通过static保持线程同步(虽然static是加在静态属性上的，但原理相同)

* + - * 1. 修饰类的class对象

给class加锁和上例的给静态方法加锁是一样的，所有对象公用一把锁

class MyThread implements Runnable {

public void test() {  
 synchronized (MyThread.class) {  
 System.*out*.println("test开始..");  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("test结束..");  
 }  
   
 public void run() {  
 this.test();  
 }  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 3; i++) {  
 Runnable runnable = new MyThread();  
 new Thread(runnable).start();  
 }  
 }  
}

因此：

**1.无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是实例对象锁，不同对象即使同属于一个类，他们的同步锁都是互不干扰的；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是类锁，是该类所有对象共享的同一把锁，只要同属一个类，即使是不同对象，他们的锁都是共享的。**

**2.** **当一个线程访问对象的一个synchronized同步代码块时，另一个线程仍然可以访问该对象中的非synchronized同步代码块。如果存在多个synchronized同步代码块，即使两个线程试图访问的不是一个同步代码块，但依然会受锁限制，因为此时锁住的是实例对象或者类**

**3.实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制，即被锁住的代码越少越好，尽量使用代码块加锁，避免在方法上直接加锁。**

这里有一个经典的比方帮助理解：

一个object就像一个大房子，大门永远打开。房子里有很多房间（也就是方法）。这些房间有上锁的（synchronized方法）， 和不上锁之分（普通方法）。房子客厅的桌上放着一把钥匙（key），这把钥匙可以打开所有上锁的房间，但钥匙只有一把。所有想调用该对象方法的线程都是想进入某个房间的人。注意一点，每个人每次使用完上锁的房间后会马上把钥匙还回去，即使他要连续使用两间上锁的房间，中间他也要把钥匙还回去，再取回来。

一个人a想进入某间上了锁的房间A，正好此时桌上有钥匙(说明暂时还没有其他人要使用上锁的房间)，于是他拿走了钥匙，虽然他拿走了钥匙，但这时其他人仍可以不受限制的使用那些不上锁的房间，但当b想使用上锁的房间B，发现钥匙没了，只能等待a还回钥匙。可能a还回钥匙时，有多个人等待拿走钥匙，其中一部分人喊着自己比较急(优先级高)，但具体钥匙给谁，由JVM决定。如果某个人在使用房间时，觉得房间床很软，想睡一会儿(sleep)，那么其他人也只能等着他醒来继续使用房间，如果某个人在使用房间时，接到指令，要等一下(wait) ，那么它必须交出手上的钥匙，老老实实地等着

* + - 1. Synchronized的底层实现

Synchronized是一个重量级的锁，它强制多个线程串行执行，丧失了多线程的性能优势。这虽然可以解决线程安全问题，但在多数情况下，锁竞争并不激烈，为了保证线程安全我们还是需要使用synchronized关键字，但这大大降低了并发的效率。在jdk1.6以前，我们通常是通过Lock框架来实现高性能锁的，但自jdk1.6起，官方对synchronized加入了很多优化措施，如自适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等。现在synchronized的性能已经不比Lock差了，但在一些灵活性要求高的场景下，还是需要使用Lock接口。

从JAVA6开始，synchronized锁分为三种状态：偏向锁，轻量级锁和重量级锁状态，这几个状态会随着线程竞争情况逐渐升级。偏向锁和轻量级锁都是乐观锁，而重量级锁是悲观锁。

当代码块中加了关键字Synchronized，编译后相应的字节码会在对应位置加上monitorenter和monitorexit指令，分别对应的是加锁操作和解锁操作；当修饰方法时，编译器会生成ACC\_SYNCHRONIZED进行标识，实际上也是隐式地加了monitorenter和monitorexit指令，当代码执行到这两个指令时，会根据对象头中锁对应的形态选择/升级锁类型(偏向锁、轻量级锁、重量级锁)

* + - * 1. 偏向锁

我们都知道，对一个类的非静态方法进行synchronized声明，在类的实例对象执行到该方法块时，会获取实例对象的对象锁，那么对象锁是在哪标记的呢？答案是在对象的对象头中。

HotSpot虚拟机中，对象在内存中分为三个部分：

1. **对象头：**对象头又分为两部分：
2. **Mark Word:** 用于存储对象自身的运行时数据，如HashCode，GC分代年龄等，这部分数据长度在32位和64位虚拟机中分别为32bit和64bit，它是实现锁的关键
3. **Klass Point:** 指向方法区对象类型数据的指针，如果是数组的话，还有一个额外的空间储存数组长度。虚拟机通过这个指针来确定对象是哪个类的实例
4. **实例数据：**主要用于存放对象的数据信息
5. **对齐填充：**这一部分不一定存在，如果对象头+实例数据不是8 byte的整数倍时，就需要对齐填充将其填满。因此空对象的大小是8 byte，其中大部分都是对齐填充

下图是64位HotSpot的MarkWord中五种状态对应的bit位使用情况



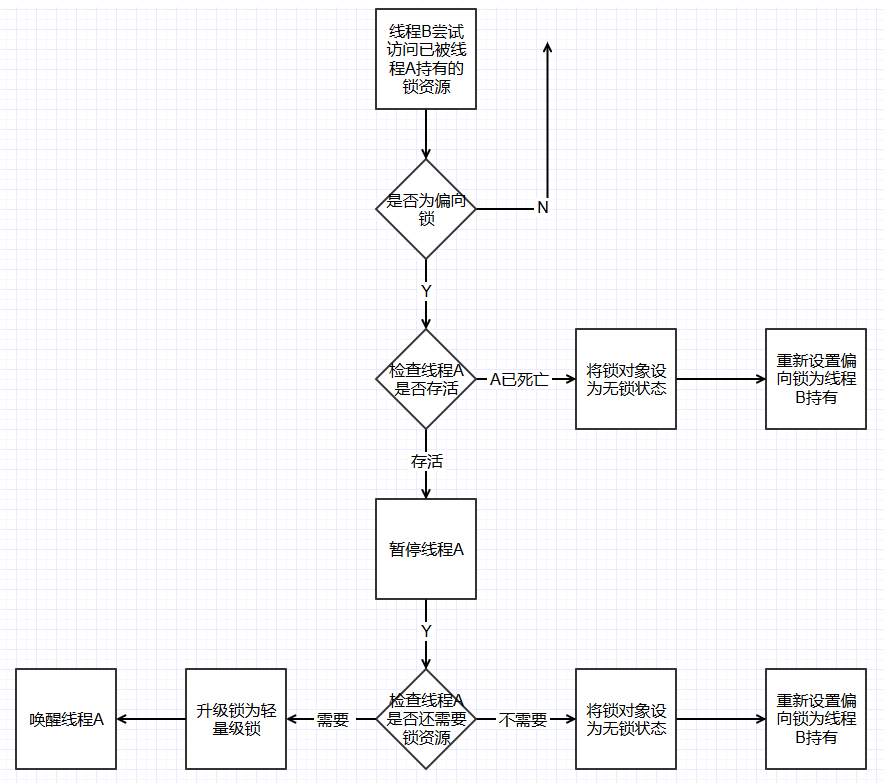
当线程A访问锁资源时，如果该锁并未被任何线程持有(无锁状态)，那么将在MarkWord中通过CAS操作存储线程A的ID和时间戳Epoch(真实地生成一个时间戳代价还是蛮大的，因此这里应当理解为一种类似时间戳的id)，同时将偏向锁标识设为1。此后，线程A如果再次访问锁资源，只需要比较当前线程的threadID和MarkWord中的threadID是否一致，如果一致，则无需使用CAS来加锁、解锁(可重入)

如果线程B尝试访问该锁资源，首先判断当前锁的MarkWord中是否标记为偏向锁。这里由于线程A已获取偏向锁，接下来线程B将判断拥有偏向锁的线程A是否还存活

如果线程A不存活，重新设置偏向锁标记为0**(线程A执行完毕后，不会主动去释放偏向锁，等到竞争出现才释放锁)**，接着使用cas替换偏向锁threadID为线程B的ID，锁不升级，仍为偏向锁

如果线程A还存活，首先暂停线程A，马上执行线程A的操作栈，检查锁的使用情况，如果线程A仍然需要持有偏向锁，则偏向锁升级为轻量级锁**（偏向锁就是这个时候升级为轻量级锁的）**，最后唤醒线程A。如果线程A不需要使用了，则将实例对象恢复成无锁状态，然后重新偏向线程B。

偏向锁假定将来只有一个线程会申请锁，且只有一个线程使用锁的情况下，避免使用轻量级锁所带来的性能开销

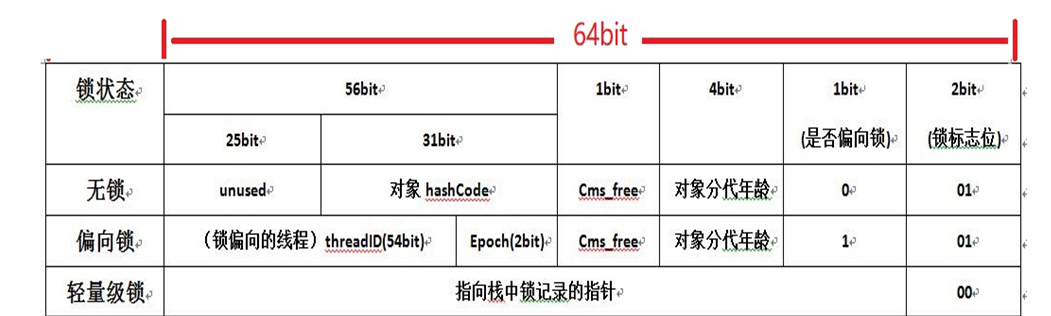


* + - * 1. 轻量级锁

上节我们了解到，当有别的线程来争用偏向锁，偏向锁将膨胀为轻量级锁。轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。试想一下，如果出现上述场景，传统的重量级synchronzied，其线程上下文切换的时间开销将是巨大的，甚至有可能比线程持有锁的时间还长，在这种情况下，我们需要轻量级锁通过CAS的实现方式，避免这种上下文切换的开销

传统的重量级锁，基于互斥信号量实现，拿到互斥信号量即为拿到锁，接着就可以对锁资源进行访问。如果没有拿到互斥信号量，线程会被挂起。

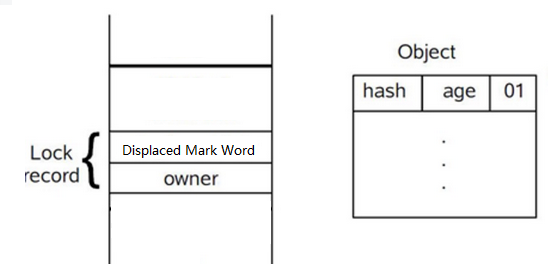
而原子的CAS操作通过compare对比当前值和预期原来值是否相同，如果相同，则认为共享资源中间没有被修改过(就好像拿到了锁一样)，可以访问共享资源。如果不相同，线程不会被挂起，一般会自旋不断重新尝试，如果锁被别的线程持有的时间短，那么CAS操作很快将成功，避免了上下文切换的开销



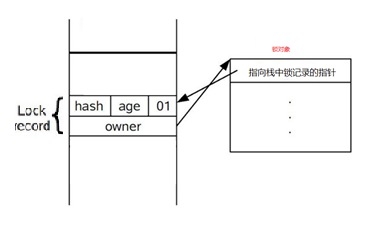
加锁过程:

首先在当前线程的栈桢中创建一块lock record空间，这个lock record分为两个部分：

1. **Displaced Mark Word：**用于存储mark word的拷贝
2. **Owner：**指向锁对象的mark word，用于识别被锁住的对象是哪一个



接着拷贝mark word到lock record空间内，接下来的CAS操作要用到它。接着使用CAS操作将上上张图中 ”指向栈中锁记录的指针” 的值替换为lock record的地址，如果cas操作成功，代表轻量级锁加锁成功，再将Lock record里的owner指针指向锁对象的mark word，如下图：

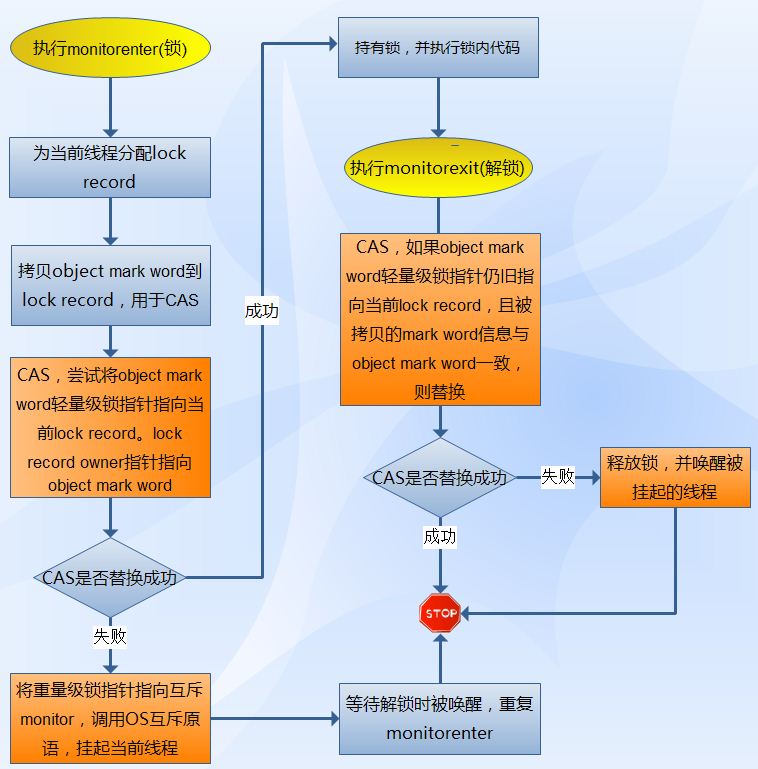


如果操作失败，尝试自旋重新操作，当自旋超过一定的次数，或者一个线程在持有锁，一个在自旋，又有第三个线程访问时，说明当前场景不适合轻量级锁，将膨胀为重量级锁

在上例中，A从偏向锁升级为轻量级锁，B在自旋，如果B自旋超过一定次数，或者线程C来了，就会升级成重量级锁

如果CAS替换线程成功，则代表获取轻量级锁，接着执行锁内代码，当执行完毕释放锁时，使用CAS将Displaced Mark Word中存储的拷贝替换回到锁对象的mark word中，若替换成功，则解锁完成。若替换不成功，表示在当前线程持有锁的这段时间内，其他线程也竞争过锁，并且锁已经膨胀成了重量锁(重量级锁就是传统的synchronzied)，那么就要去唤醒一个被挂起的线程去重新竞争锁

Ps：其实只要出现了锁资源访问，都会在对应的线程栈中生成一个Lock Record，查阅源码后发现偏向锁其实也是有Lock Record的，偏向锁和轻量级锁不像重量级锁有\_recursions属性，其可重入性是依靠统计线程栈中Lock Record个数来记录的，每次重入都会生成一个Lock Record，



* + - * 1. 重量级锁

Java6以前，synchronized只有重量级锁一种模式

我们都知道重量级锁一旦出现，意味着必然有线程没拿到锁，需要阻塞并等待持有锁的线程释放锁后唤醒。那么就有一个问题，需要被唤醒的多个线程要存储在哪呢？显然对于重量级锁，Lock Record不再满足要求，因此重量级锁引入了ObjectMonitor类，这个类是一个由c++实现的类



上图中”指向重量级锁的指针”就是指向ObjectMonitor对象的指针，ObjectMonitor对象和java对象一一对应，源码如下：

#ObjectMonitor.hpp  
 ObjectMonitor(){  
 //记录原先的MarkWord，一般是轻量级锁状态下的MarkWord  
 \_header = NULL;  
 \_count = 0;  
 //等待锁的线程个数  
 \_waiters = 0;  
 //线程重入次数  
 \_recursions = 0;  
 //一一对应的java对象  
 \_object = NULL;  
 //占有当前锁的线程，会指向占有锁的线程的lock record  
 \_owner = NULL;  
 //调用wait方法后等待锁的队列，也就是我们前面说过的[等待池](#锁池和等待池)  
 \_WaitSet = NULL;  
 //等待队列的锁  
 \_WaitSetLock = 0;  
 \_Responseable = NULL;  
 \_succ = NULL;  
 //   
 \_cxq = NULL;  
 FreeNext = NULL;  
 //获取锁失败后，存放被阻塞线程的队列，也就是我们前面说过的[锁池](#锁池和等待池)  
 \_EntryList = NULL;  
 \_SpinFreq = 0;s  
 \_SpinClock = 0;  
 OwnerIsThread = 0;  
 \_previous\_owner\_tid = 0;  
}

重量级锁加锁时，首先会在内存中分配ObjectMonitor对象，ObjectMonitor对象中存储着当前持有锁的线程\_owner、等待池\_WaitSet、锁池\_EntryList、重入次数\_recursions

当从轻量级锁膨胀到重量级锁后，被阻塞的线程将进入锁池\_EntryList中等待竞争锁资源，当线程调用wait()/join()方法后，线程进入等待池\_WaitSet，随后如果被notify()/notifyAll()唤醒/wait()时间到/join()加入的线程执行完毕，线程将进入锁池\_EntryList，等待当前持有锁的线程\_owner执行完毕释放锁后，一起竞争锁资源

重量级锁的底层是基于操作系统的mutex互斥锁实现的，它的每一个线程都对应操作系统的一个原生线程，重量级锁如果需要阻塞或者唤醒一条线程，都需要操作系统来帮忙实现，这就需要从用户态切换到内核态，这个操作是非常耗时的

* + - * 1. 自适应自旋

在很多情况下，对象锁的锁定状态只会持续很短的一段时间，例如整数的自加操作，为这么短的锁定时间增加上下文切换的开销明显不值得，为此引入了自旋锁。

所谓“自旋”，就是让线程去执行一个循环，在循环中不断竞争锁，如果竞争不到继续循环，直到成功获取锁资源。在自旋的过程中，线程一直处于运行(running) 状态，免去了上下文切换带来的开销，但长时间自旋就变成了“忙碌式等待”，因此一般会通过JVM参数给定一个最大自旋次数，超出次数后，将挂起线程

但上述自旋锁还不能满足要求，JDK1.6将自旋锁改为”自适应自旋锁”，这意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定：

1.如果在同一个锁对象上，自旋等待刚刚成功获得过锁，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许本次自旋等待持续相对更长的时间，比如100个循环。

2.相反的，如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能减少自旋时间甚至省略自旋过程，以避免浪费处理器资源。

最典型的例子就是上面轻量级锁升级为重量级锁中会采取自适应自旋

* + - * 1. 锁消除和锁粗化

**锁消除：**锁消除是编译器级别的事情。如果编译器发现不可能被共享的对象，则可以消除这些对象的锁操作。

在JDK实现中存在一些锁，可能在一些场景下我们并不需要，比如Vector和StringBuffer这样的类，它们中的很多方法都是有锁的。当我们在一些不会有线程安全的情况下使用这些类的方法时，达到某些条件时，编译器会将锁消除来提高性能。

**锁粗化：**通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完锁资源后，应该立即释放锁。只有这样，等待在这个锁上的其他线程才能尽早的获得资源执行任务。

但是，凡事都有一个度，如果对**同一个锁**不停的进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化 。

如：

public void demoMethod(){ public void demoMethod(){

synchronized(lock){ synchronized(lock){  
 //do sth1. //do sth1.

} //do sth2.

synchronized(lock){ }

//do sth2. }  
 }   
}

* + 1. volatile和synchronized的区别及使用场景

#### 6.4.7.1 二者的区别

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。

二者的区别如下：

1.volatile轻量级，开销小(几乎可以忽略不计)，不会造成阻塞，只能修饰类变量。synchronized重量级，开销大，会造成阻塞，只能修饰方法和代码块

2.volatile只能保证数据的可见性和一定程度上的有序性，不能用来同步，synchronized三种特性都能保证，可以用于线程同步

synchronized不仅保证可见性，而且还保证原子性，因为，只有获得了锁的线程才能进入临界区，从而保证临界区中的所有语句都全部执行。多个线程争抢synchronized锁对象时，会出现阻塞。

#### 6.4.7.2 volatile的使用场景

通常来说，使用volatile必须具备以下2个条件：

**1.对变量的写操作不依赖于前值，**如n=n+1,n++等都依赖于前值，这使得volatile 变量不能用作线程安全的计数器

**2.该变量没有包含在具有其他变量的不变式中，即如果有多个volatile变量，则每个volatile变量必须独立于其他的volatile变量，一个表达式中不能同时含有两个或两个以上的volatile变量。**

一般情况下，在并发环境下需要同时满足上面的两个原则会比较难，这也决定了volatile的使用场景非常有限。如果多线程执行的操作不是原子性的，使用volatile时就一定要慎重(尽量避免使用)。

* 1. ThreadLocal

### 6.5.1 线程封闭技术

当访问线程不安全的共享数据时，我们往往采用同步保证其线程安全，但实现线程同步是一件困难的事情，并且同步机制会有一定的性能损耗。对于一些特殊场景，我们可以考虑采用线程封闭技术来保证线程安全

那么什么是线程封闭技术呢？简单地来说，线程封闭技术就是不共享数据，将线程不安全的共享数据封闭在各自的线程中，在释放该数据前，不允许其他线程访问。这样一来，即使这个数据是线程不安全的，也不会再出现线程安全问题

线程封闭技术的一个常见的应用场景是JDBC。JDBC中的Connection对象一般是存放在数据库连接池中由多个线程共享的。在web应用程序中，web应用线程从连接池获取一个Connection对象，使用完之后将对象返还给连接池，使用Connection对象时，该对象被封闭在这个web应用线程中，在Connection对象被还给连接池之前，数据库连接池不会再将其分配给其他线程使用

### 6.5.2 什么是ThreadLocal

从JDK1.2起，java就为我们提供了Java.lang.ThreadLocal，ThreadLocal使用了线程封闭技术帮助我们维护线程的局部变量，它为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

一个ThreadLocal只代表一个变量，它维护的是一个变量在多个线程之间的副本

**ThreadLocal 并不解决多线程变量共享的问题，ThreadLocal 适用于每个线程需要使用自己独立的变量且该变量需要在多个方法中被使用，也即变量在各线程间隔离而在同一线程使用的方法或类间共享的场景，如：**



ThreadLocal 变量通常被private static修饰。当一个线程结束时，它所使用的所有 ThreadLocal 相对应的变量副本都会被回收

class ThreadLocalTest {

private static final ThreadLocal<Integer> *local* = ThreadLocal.*withInitial*(()->0);  
 public static void main(String[] args) {  
 for(int i = 0; i < 5;i ++){  
 new TestThread(*local*).start();  
 }  
 }  
}  
class TestThread extends Thread{  
 //作为类的属性是希望可以被多个方法共用，但又同时希望保证其线程安全

private static ThreadLocal<Integer> *local*;  
  
 public TestThread(ThreadLocal local){  
 this.*local*=local;  
 }  
 @Override  
 public void run() {  
 System.*out*.println(Thread.*currentThread*().getName() + " start, local = " + *local*.get());  
 for(int i = 0; i <= 10;i++){  
 *local*.set(*local*.get() + i); //计算(end+1)\*end/2的值  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + " end, local = " + *local*.get());  
 }  
}

输出结果：

Thread-1 start, local = 0

Thread-0 start, local = 0

Thread-0 end, local = 55

Thread-2 start, local = 0

Thread-1 end, local = 55

Thread-2 end, local = 55

Thread-3 start, local = 0

Thread-4 start, local = 0

Thread-3 end, local = 55

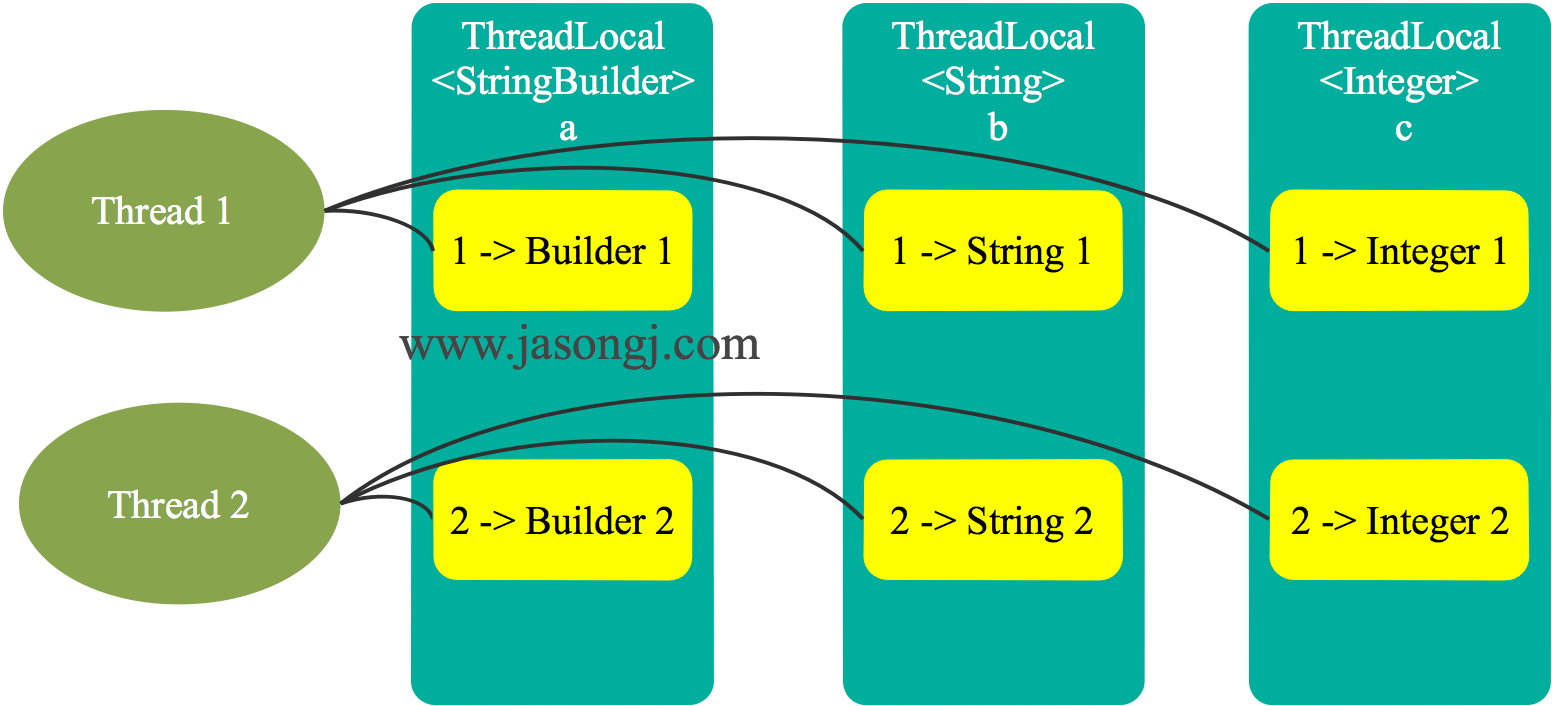
Thread-4 end, local = 55

每个线程最后的结果都是55，这表示local在每一个线程中的值互不干扰

### 6.5.2 早期ThreadLocal原理

起先ThreadLocal是这么设计的：

每个ThreadLocal类的内部创建一个Map，然后用线程的ID作为Map的key，该线程对应的变量副本作为Map的value，这样就能达到各个线程的值隔离的效果。下图是三个ThreadLocal，因此有三个map，维护三个变量a、b、c在两条线程中的6个副本，两个线程向这三个ThreadLocal各自维护的map中存入一个变量(由于是两个线程，因此有两个副本)



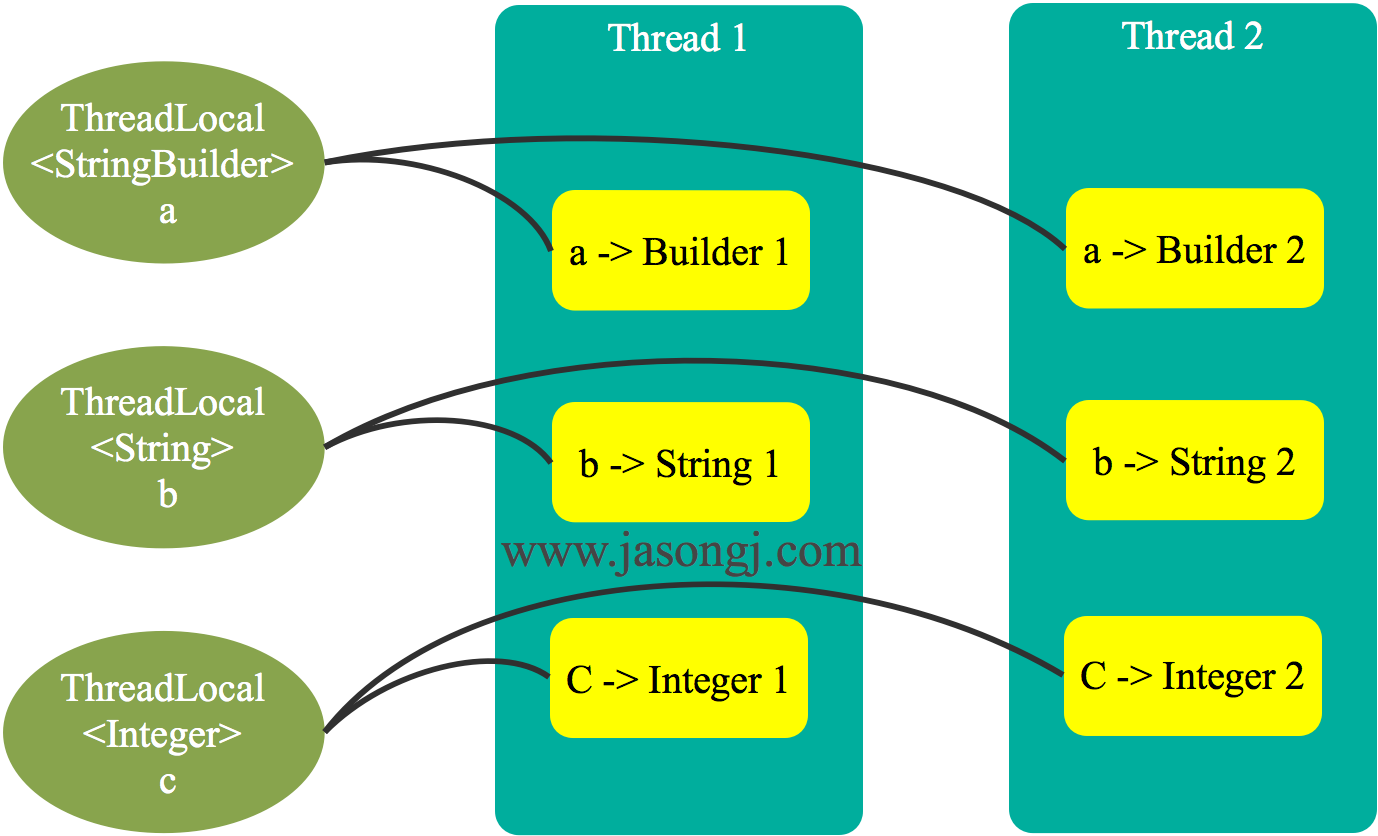
看起来一切都那么完美，我们做到了在多个线程之间提供多个变量副本以隔离变量的目的，每当有新线程访问该 ThreadLocal 时，需要向 Map 中添加一个映射(为新线程提供副本)，而每个线程结束时，应该清除该线程的变量副本映射。但是，有两个严重的问题急需解决

1.增加线程与减少线程均需要写操作ThreadLocal的Map，故需保证该 Map 线程安全。

2. 线程结束时，需要保证它所访问的所有 ThreadLocal 中对应的映射均删除，否则可能会引起内存泄漏。

### 6.5.3 ThreadLocal原理

没过多久，jdk就换了一种ThreadLocal实现方式，在新的实现方式中，Map不再由ThreadLocal维护，而是**每个Thread维护一个ThreadLocalMap映射表，这个映射表的key是ThreadLocal实例本身，value是真正需要存储的副本变量。**



之前由于变量的副本不在线程自己的作用域内维护，而是把多个线程的变量统一交由ThreadLocal维护，这导致需要加锁来保证多线程写的问题。现在，Map 由 Thread 维护，从而使得每个 Thread 只访问自己的 Map，彻底避免了加锁

Java的Thread类中提供了一个ThreadLocal.ThreadLocalMap类，通过ThreadLocal的getMap()方法即可获得当前线程的ThreadLocal.ThreadLocalMap对象。然后把threadLocal对象作为key，通过ThreadLocal.ThreadLocalMap对象即可获得当前线程的变量副本

### 6.5.4 ThreadLocal使用及源码分析

ThreadLocal支持泛型创建，主要使用四个方法：get()，set()，initialValue()，remove()

一般来说是先set再get才能取到值，如果先调用get方法，并且之前并没有调用set方法，那么get方法会调用initialValue方法来初始化一个值，因此这种情形下必须重写initialValue方法

下面是这四个方法的源码：

public T get() {

Thread t = Thread.*currentThread*();//获取当前使用ThreadLocal的线程  
 ThreadLocalMap map = getMap(t); //getMap方法就一句：return t.threadLocals;由于Thread类和ThreadLocal都是java.lang包下的，且Thread类的threadLocals属性访问权限被设为default，因此可以直接访问  
 if (map != null) {  
 ThreadLocalMap.Entry e = map.getEntry(this); //map不空，获取当前线程的ThreadLocal存储的变量副本  
 if (e != null) {  
 @SuppressWarnings("unchecked")  
 T result = (T)e.value;  
 return result;  
 }  
 }

//如果ThreadLocalMap为null，就给当前线程创建一个ThreadLocalMap并初始化一个value加入到ThreadLocalMap中，然后将value返回，由于setInitialValue内部使用了initialValue方法，因此返回值value是initialValue方法的返回值  
 return setInitialValue();  
}

public void set(T value) {  
 Thread t = Thread.*currentThread*();//获取当前使用ThreadLocal的线程  
 ThreadLocalMap map = getMap(t); //获取ThreadLocalMap  
 if (map != null)  
 map.set(this, value);  
 else  
 createMap(t, value); //如果map为空就创建一个  
}

protected T initialValue() {//在先调用get再set的情形下，重写该方法来为ThreadLocal赋初始值  
 return null;  
}

public void remove() {//移除当前线程对应于本ThreadLocal的副本，其实该副本是存储在当前线程的ThreadLocalMap中的  
 ThreadLocalMap m = getMap(Thread.*currentThread*());  
 if (m != null)  
 m.remove(this);  
}

* 1. CAS技术

CAS（Compare and Swap），即比较并替换

在6.4中，我们了解到在操作共享资源时，会出现三个问题：有序性，可见性和原子性。一般情况下，我们采用synchronized同步锁(独占锁、互斥锁)来解决这个问题，即同一时间只有一个线程能够修改共享变量，其他线程必须等待。但是这样的话就相当于单线程，体现不出来多线程的优势。那么我们有没有其他方式来解决这个问题呢？答案是肯定的，这就是我们的CAS技术，在学习CAS技术之前，我们先了解两个重要概念：悲观锁与乐观锁。

### 6.6.1 乐观锁和悲观锁

和数据库锁一样，JAVA在多线程中也有乐观锁和悲观锁

**悲观锁:** 悲观地假定会发生并发冲突。所以当某个线程获取共享资源时，会阻止别的线程获取共享资源。也称独占锁或者互斥锁，例如java中的synchronized同步锁就是悲观锁。

**乐观锁:** 乐观地假设不会发生并发冲突，只有在最后更新共享资源的时候会判断一下在此期间有没有别的线程修改了这个共享资源。如果发生冲突就重试，直到没有冲突，更新成功。乐观锁是一种思想，而CAS就是这种思想的实现方式之一。

悲观锁会阻塞其他线程。乐观锁不会阻塞其他线程，如果发生冲突，采用死循环的方式一直重试，直到更新成功。

### 6.6.2 CAS技术

#### 6.6.2.1 基本原理

CAS的原理很简单，包含三个值**内存位置(V)**、**预期原来值(A)**以及**期待更新值(B)**。

如果**内存位置(V)**存放的值和**预期原来值(A)**相同，那么就将该变量的值更新为**期待更新值(B)**，这个过程就是比较并替换(Compare And Swap)，即CAS

但是这样一来，存在一个问题，比如当前线程比较V和A后，发现二者相等，正准备更新共享变量值的时候，这个共享变量值被其他线程更改了。这是因为整个CAS的过程不是原子性操作所导致的，那么JAVA是如何解决这个问题的呢？

在很早的时候，诸多硬件厂商就在芯片中植入了并发操作的原语，但遗憾的是，早期的java并不能够利用硬件提供的这些便利来提升系统的性能，直至Java本地方法(JNI，Java Native Interface)的出现，它为java程序越过JVM直接调用本地方法提供了一种便捷的方式，至于JNI，这里就不在过多赘述，只需要知道，至此JDK中使用的CAS成为了CPU中的一条原子指令，不会再出现上述数据不一致的问题

扩展知识：jdk中sun.misc.Unsafe类提供的方法可以像C语言指针一样直接操作内存，这是实现CAS原子性操作的关键，Unsafe中的方法大量使用native声明，这些native声明的方法并不提供实现体（有些像定义一个Java Interface），因为其实现体是由非Java语言在外面实现的。Unsafe类正是通过这些方法调用由C实现的操作CPU原语的函数来保证CAS的原子性的，不过，不建议使用Unsafe类，从该类的命名就可以看出这不是一个安全的类

当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。但和synchronized不同，失败的线程并不会被挂起，而是仅是被告知失败，并且允许其再次循环尝试(发生争用时不会将线程挂起，因此也不会有上下文切换的开销)，当然也允许其直接放弃操作共享变量。整个过程不仅保证了对共享变量的有序访问，并且由于其中没有锁的存在，也不会出现死锁。因此我们可以说**CAS操作天生免疫死锁**

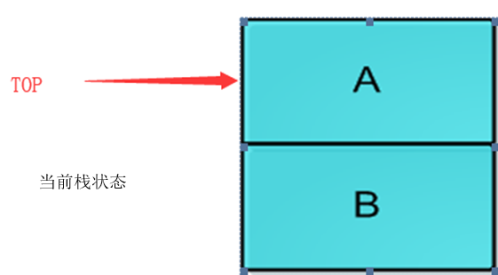
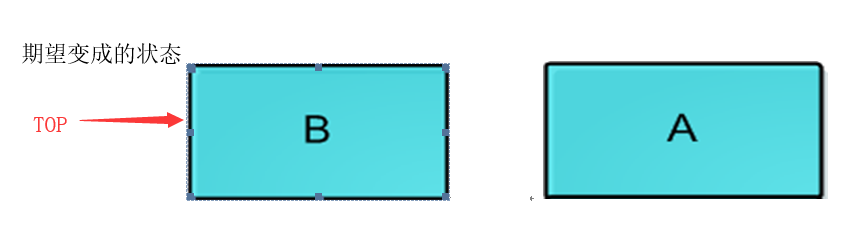
**原子操作一定是线程安全的，即使使用的cpu是多核的。在单核CPU中, 能够在一个指令中完成的操作都可以看作为原子操作, 因为中断只发生在指令间。在多核cpu中，即使是单个指令，也可能受到别的CPU核心的干扰，比如被原子操作保证后的i++，，如果多个CPU运行的多个进程在同时对这一块内存执行这个指令，结果依然是无法预料的。为了解决这个问题，x86体系提供了提供了硬件层面的解决方案，以牺牲部分性能为代价，保证了多核CPU的原子操作的线程安全**

虽然通过CPU原语解决了原子性问题，但CAS仍然会来带一些问题

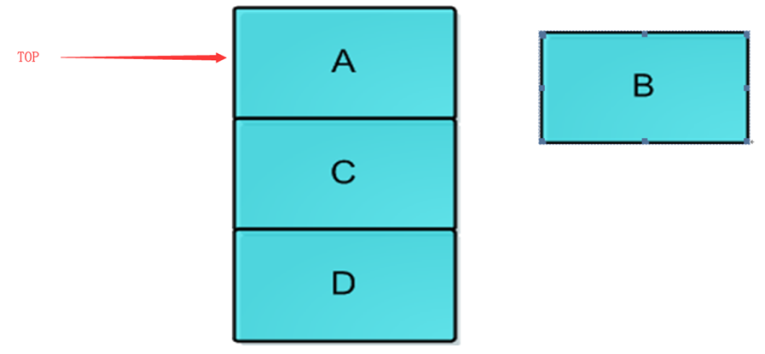
#### 6.6.2.2 ABA问题

CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，后来变成了B，又重新变回了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。如下：

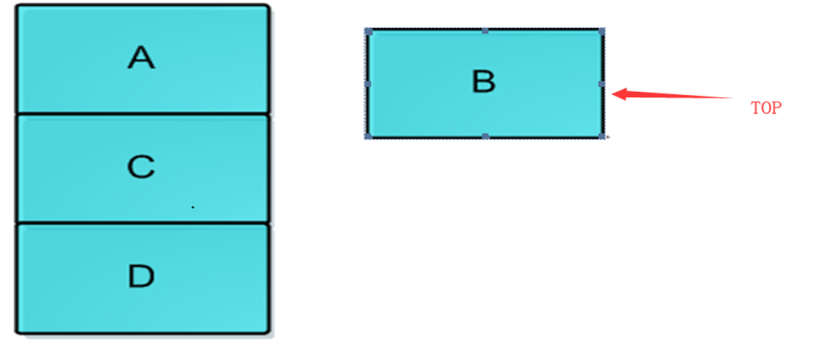
假设现有一个栈，栈顶元素是A，并且A.next=B，我们希望把栈顶元素设为B，即A出栈。对于这样一个步骤，我们首先调用CAS将top指向B，在CAS操作中top的原值为A，当前变量值也为A，期望更新值为B。接着再将A.next设为null。

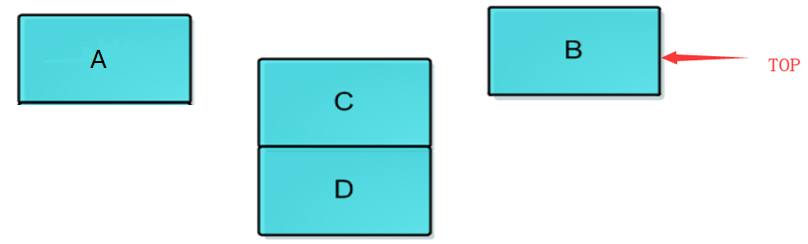
在CAS已获取三个参数，但还未执行操作前，另一个线程介入，pop A、B，然后pushD、C、A，此时B已不在栈内，B.next为null。但对于第一个线程，top的原值为A，当前变量值也为A，期望更新值仍然是B，即：



然后第一个线程进行CAS操作，比较成功，将栈顶元素置换为B，top指向B，但B.next为null，并不指向C元素。



CAS结束后，再执行A.next=null，就变成了这个样子，这显然不符合我们的预期(由于第二个线程的介入，我们的预期变为B作为栈顶，B.next=C，C.next=D)：



对于这种问题，解决的思路是对变量使用版本号来解决。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B-3A

从java1.5开始，jdk提供了一个带有时间戳的对象引用—AtomicStampedReference原子类。在每次修改后，AtomicStampedReference不仅会设置新值而且还会记录更改的时间。当AtomicStampedReference设置要更新的对象值时，当前对象值以及时间戳都必须同时满足原值才能写入成功，这也就解决了ABA问题

#### 6.6.2.3 其他问题

除了ABA问题外，CAS还有一些其他问题：

**1.循环开销时间大：**如果线程CAS操作失败，会一直进行尝试。如果CAS长时间一直不成功，可能会给CPU带来很大的开销。

**2.只能保证一个共享变量的原子操作：**当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作。但是由于底层封装的CPU指令一次只能原子地修改一个变量，对多个共享变量操作时，循环CAS就无法保证操作的原子性。有一个解决方案是利用对象整合多个共享变量，即一个类中的成员变量就是这几个共享变量。然后将这个对象做CAS操作就可以保证其原子性。atomic中提供了原子引用AtomicReference(见6.6.3.2)来保证引用对象之间的原子性

#### 6.6.2.4 自旋锁

当线程获取锁时，如果获取失败，一般有两种操作，一种是阻塞当前线程，synchronizied就是这么做的，另一种是获取锁失败的线程将不断循环尝试获取锁，这就是自旋锁，是CAS操作失败的常见选择

如果一个线程希望访问被自旋锁保护的共享资源，如果尝试获取锁时，没有其他线程持有锁，那么该线程立即持有自旋锁；如果尝试获取锁时，锁已被其他线程持有，那么该线程将不断自旋，直至其他线程释放锁资源

在轻度到中度的争用情况下，非阻塞算法的性能会超越阻塞算法，因为 CAS 的多数时间都在第一次尝试时就成功，而发生争用时的开销也不涉及线程挂起和上下文切换，只多了几个循环迭代。没有争用的 CAS 要比没有争用的锁便宜得多，但如果线程执行的任务需要非常长的时间，或者线程对共享数据的竞争相当激烈,那么使用自旋锁的效率就很低。它会使等待的线程一直处于死循环状态，白白浪费CPU

### 6.6.3 原子类

通过上面我们可以知道，java.util.concurrent是依赖CAS操作实现的。JUC还为我们提供了另外一个工具包，原子类包java.util.concurrent.atomic，通过这个包，我们可以在单个变量上实现原子性操作，当然了，原子类的原子性操作也是依赖于CAS技术实现的

java.util.concurrent.atomic一共提供了12个原子类，分为下面四组：

**AtomicInteger和AtomicIntegerArray：基于Integer类型**

**AtomicBoolean：基于Boolean类型**

**AtomicLong和AtomicLongArray：基于Long类型**

**AtomicReference和AtomicReferenceArray：基于引用类型**

AtomicBoolean和AtomicLong的使用和内部实现原理几乎和AtomicInteger一样，因此这里我们重点讲解AtomicInteger

#### 6.6.3.1 AtomicInteger原子整型

首先是构造方法：

private volatile int value;

public AtomicInteger(int initialValue) {

value = initialValue;  
}public AtomicInteger() {}

我们可以通过指定一个int值的方式来通过构造函数初始化AtomicInteger，，如果调用了无参的构造函数，那么AtomicInteger将会初始化为0

当然，AtomicInteger也提供了set()和get()方法来帮助我们设置/获取int值，但需要注意的是，**这两个操作都是非原子性的**：

public final int get() {

return value; //包含两个操作。1.读取value的值；2.返回value的值   
}public final void set(int newValue) {  
 value = newValue; //包含两个操作。1.读取newValue的值；2.将 newValue的值赋值给value  
}

如果希望通过原子性操作设置值，应当使用下面的方法：

public final int getAndSet(int newValue) {

//原子地设置新值并返回原来的值，该方法设置值操作是原子的，但返回原值的操作不是原子的

return *unsafe*.getAndSetInt(this, *valueOffset*, newValue);  
}

下面列出所有AtomicInteger的常用原子方法：

public final int getAndSet(int newValue) //给AtomicInteger设置newValue并返回oldValue

public final boolean compareAndSet(int expect, int update) //如果当前的AtomicInteger值和期望值expect相等就set并返回true/false，即CAS操作  
public final int getAndIncrement() //对AtomicInteger原子的加1并返回当前自增前的value  
public final int getAndDecrement() //对AtomicInteger原子的减1并返回自减之前的的value  
public final int getAndAdd(int delta) //对AtomicInteger原子的加上delta值并返加之前的value  
public final int incrementAndGet() //对AtomicInteger原子的加1并返回加1后的值  
public final int decrementAndGet() //对AtomicInteger原子的减1并返回减1后的值  
public final int addAndGet(int delta) //给AtomicInteger原子的加上指定的delta值并返回加后的值

#### 6.6.3.2 AtomicReference <V>原子引用

AtomicReference使用了泛型，首先是构造方法，如果调用无参构造函数，其默认值为null：

private volatile V value;

public AtomicReference(V initialValue) {  
 value = initialValue;  
}public AtomicReference() {  
}

同样的，其get和set方法都不是原子性的：

public final V get() {

return value; //包含两个操作。1.读取value的值；2.返回value的值   
}public final void set(V newValue) {  
 value = newValue; //包含两个操作。1.读取newValue的值；2.将 newValue的值赋值给value  
}

下面列出其常用原子方法：

public final V getAndSet(V newValue); //给AtomicReference设置newValue并返回oldValue

public final boolean compareAndSet(V expect, V update); //如果当前的AtomicReference值和期望值expect相等就set并返回true/false，即CAS操作

* 1. Lock框架

从jdk1.5开始，java提供了另外一种方式来实现同步访问，那就是Lock框架，那么既然已经有了synchronized了，为什么还要使用Lock框架？这是因为synchronized关键字有极大的缺陷(jdk1.6以前)

* + 1. synchronized的缺陷

如果一个代码块被synchronized修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，直到获取锁的线程释放锁，而这里获取锁的线程释放锁会有三种情况：

　 1.获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

　　2.获取锁的线程执行发生异常，此时JVM会让线程自动释放锁；

3.获取锁的线程执行遇到join()或wait()方法，在等待的时候立即释放锁

如果已获取锁的线程由于要等待IO或者其他原因（比如调用sleep方法）被阻塞了，但是又没有释放锁，其他线程便只能”眼巴巴”等着，这严重影响多线程程序的并发执行效率。因此就需要有一种机制可以不让等待的线程一直无期限地等待下去（比如只等待一定的时间或者能够响应中断），通过Lock就可以办到。

比如另一个场景：当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。如果采用synchronized关键字来实现同步的话，即使多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

Synchronized无法得知线程是否成功获得锁资源

Lock提供了比synchronized更多的功能。但是要注意以下几点：

1.synchronized是Java的关键字，因此是Java的内置特性，是基于JVM层面实现的。而Lock是一个Java接口，是基于JDK层面实现的，通过这个接口可以实现同步访问；

2.采用synchronized方式不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而 **Lock则必须要用户去手动释放锁，即使发生异常也不会自动释放锁**，如果没有主动释放锁，就有可能导致死锁现象。

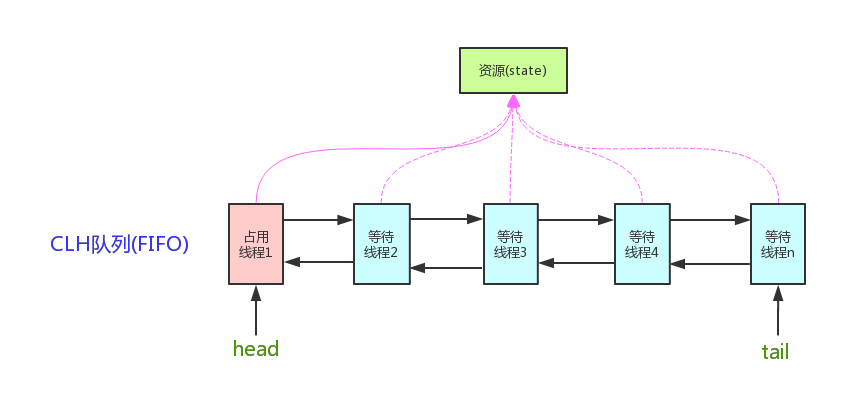
3. 在资源竞争不是很激烈的情况下，Synchronized的性能要优于Lock，但是在资源竞争很激烈的情况下，Synchronized的性能会下降几十倍

**但在JDK1.6开始，Synchronized就进行了相关优化，其性能已经不比Lock差了，但如果考虑到加锁的灵活性，还是可以使用Lock框架**

* + 1. AQS(AbstractQueuedSynchronizer)

#### 6.7.2.1 状态变量state

在学习Lock之前，我们先了解什么是AQS，AQS是jdk提供的一个同步抽象类，其中维护了一个CLH同步队列，并提供了大量的同步操作，用于实现各种同步类，我们可以通过这个抽象类实现自己的锁，Lock中的相关锁(如ReentrantLock、 ReadWriteLock等)都是通过继承AQS来构建的，这些子类只需要实现实现共享资源state的获取与释放方式即可，线程等待队列的维护(如获取资源失败入队/唤醒出队等)已经被AQS实现了：



//同步队列的头结点，表示当前正在使用锁资源的节点

private transient volatile Node head;  
  
//同步队列的尾结点  
private transient volatile Node tail;  
  
//同步状态  
private volatile int state;  
  
//获取同步状态  
protected final int getState() {  
 return state;  
}  
  
//设置同步状态  
protected final void setState(int newState) {  
 state = newState;  
}  
  
//以CAS方式设置同步状态  
protected final boolean compareAndSetState(int expect, int update) {  
 return unsafe.compareAndSwapInt(this, stateOffset, expect, update);  
}

AQS维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程资源被阻塞时会进入此队列），volatile关键字保证了代表共享资源的state变量在多个等待线程之间的可见性。State可以被继承了AQS的同步类使用，用于表示共享资源的任意状态，如重入锁ReentrantLock用它表示持有锁资源的线程重复获取锁的次数；信号量Semaphore用它来表示剩余可用的资源数量；FutureTask用它表示任务的状态(尚未开始、正在运行、已完成、已取消)等等。当然，实现了AQS的同步类还可以根据自身特点加入额外的状态变量，如重入锁ReentrantLock就加入了一个表示当前持有锁的线程的状态变量，用以区别尝试获取锁的线程是重入的还是竞争的

#### 6.7.2.2 获取锁和释放锁的标准模式

AQS要求子类根据同步资源的state状态来获取(acquire)和释放(release)同步资源

下面是AQS获取操作和释放操作的标准形式，了解它们有助于我们理解AQS的原理，在AQS不同的实现里，以下操作可以是独占的(如重入锁ReentrantLock)也可以是非独占的(如信号量Semaphore)。**在整个获取过程中，忽略中断的影响**。

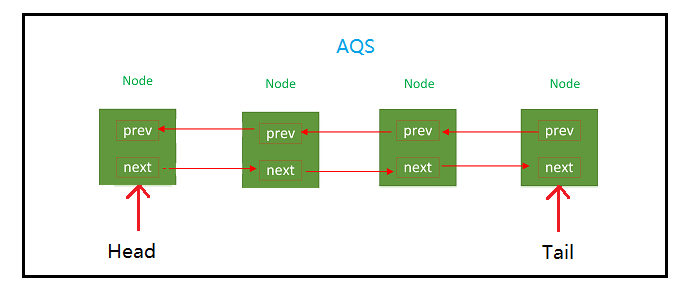
boolean acquire() throws InterruptedException {

while (获取锁失败)> {   
 if (需要阻塞当前线程的获取请求)> {  
 如果当前线程不在AQS队列中，则将其插入等待队列   
 阻塞当前残程，直到下一次循环再次尝试获取锁  
 }else{  
 直接返回false获取锁失敗  
 }  
 }  
 获取锁成功，更新同步器的state状态  
 如果线程位于等待队列中，则将其移出队列   
 返回true获取成功  
}

void release(){  
 更新同步器的state等状态变量  
 if(新的状态允许某个被阻塞线程获取成功){  
 解除等待队列中的一个或多个线程的阻塞状态  
 }   
}

#### 6.7.2.3 等待队列CLH

在上一节中我们看到，如果AQS的子类需要阻塞某个线程(让线程等待锁的释放 而不是 直接返回false表示获取锁失败)，就要将其加入等待队列中，AQS的子类中有一个内部类Node，一个内部类Node可以存放一个线程，多个Node节点组成一个FIFO队列，这是一个双向队列



CLH队列是一个非阻塞的 FIFO 队列，也就是说往里面插入或移除一个节点的时候，在并发条件下不会阻塞，这是通过自旋锁和 CAS 保证节点插入和移除的原子性，实现无锁且快速的插入。

CLH的维护已经由AQS抽象类帮我们完成了，子类无需关注队列的维护，下面就来看看CLH的入队和出队的源码

入队操作：

*/\*\**

*\* 为当前线程和给定模式创建入队节点.  
 \*/*

private Node addWaiter(Node mode) {

Node node = new Node(Thread.*currentThread*(), mode);  
 // 如果可以直接快速入队  
 Node pred = tail;  
 if (pred != null) {  
 node.prev = pred;  
 //将tail指向新Node

if (compareAndSetTail(pred, node)) {  
 //如果tail修改成功，原队列的最后一个Node的next指向新节点

pred.next = node;  
 return node;  
 }  
 }

//如果CLH需要初始化或者入队失败需要自旋  
 enq(node);  
 return node;  
}

#### 6.7.2.4 独占模式（EXCLUSIVE）

站在使用者的角度，AQS的功能主要分为两类：独占锁和共享锁。在它的所有子类中，要么实现了它的独占功能的API，要么实现了共享功能的API，但不会一个类同时使用两套API，即使是ReentrantReadWriteLock，也是通过两个内部类各自使用一套API来实现的。**AQS所有方法名中带Shared的方法全部都是共享模式需要用到的方法，不带shared的方法大多是独占模式使用的方法**

1.当AQS的子类实现**独占功能**时，就是每次只允许一个线程执行，当前线程执行完会release将同步状态归零，再唤醒后继节点。如ReentrantLock，资源是否可以被访问被定义为：只要AQS的state变量不为0，并且持有锁的线程不是当前线程，那么代表资源不可访问。

2.当AQS的子类实现**共享功能**时，只要线程数没达到限制的数量都可以直接获取资源执行，如Semaphore，资源是否可以被访问被定义为：只要AQS的state变量不为0，那么就代表资源可以访问。再比如CountDownLatch，只要AQS的state为0，被await方法阻塞的主线程就可以继续执行下去

为了判断一个结点处于何种模式，AQS的内部类Node提供了两个属性

*/\*\* 标记一个结点是共享锁模式 \*/*

static final Node *SHARED* = new Node();  
*/\*\* 标记一个结点是独占锁模式 \*/*static final Node *EXCLUSIVE* = null;

为了表示当前节点的等待状态，Node节点还提供了waitStatus属性

*/\*\* 标记一个结点是独占锁模式*

*\* CANCELLED： 值为1，在同步队列中等待的线程等待超时或被中断，需要从同步队列中取消该Node的结点，其结点的waitStatus为CANCELLED，即结束状态，进入该状态后的结点将不会再变化。*

*\* SIGNAL： 值为-1，被标识为该等待唤醒状态的后继结点，当其前继结点的线程释放了同步锁或被取消，将会通知该后继结点的线程执行。说白了，就是处于唤醒状态，只要前继结点释放锁，就会通知标识为SIGNAL状态的后继结点的线程执行。*

*\* CONDITION： 值为-2，与Condition相关，该标识的结点处于等待队列中，结点的线程等待在Condition上，当其他线程调用了Condition的signal()方法后，CONDITION状态的结点将从等待队列转移到同步队列中，等待获取同步锁。*

*\* PROPAGATE： 值为-3，与共享模式相关，在共享模式中，该状态标识结点的线程处于可运行状态。*

*\* 0状态： 值为0，代表初始化状态。*

*\*/*

volatile int waitStatus;

独占锁应当至少具有两个功能：

**1.获取锁的功能。**当多个线程一起获取锁的时候，只有一个线程能获取到锁，其他线程必须在当前位置阻塞等待。

**2.释放锁的功能。**获取锁的线程释放锁资源，而且还必须能唤醒正在等待锁资源的一个线程

##### 6.7.2.4.1获取锁

AQS通过acquire()方法获取独占锁，我们来看一下它的源码

*/\*\**

*\*获取独占模式，中途忽略中断。通过调用至少一次tryAcquire()方法实现，如果获取锁成功则返回。否则，线程将排队。*

*\*之后可能会出现反复阻塞(多次获取失败)和取消阻塞(再次尝试获取)的情况，调用tryAcquire()方法直到成功。此方法可*

*\*用于实现方法Lock()。*

*\*/*

public final void acquire(int arg) {

//尝试获取独占锁，tryAcquire由独占类型的子类实现

if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.*EXCLUSIVE*), arg))  
 *selfInterrupt*();  
}

1.执行tryAcquire()方法尝试获取锁资源，如果成功，则acquire()执行完成，即成功获取锁，否则执行下一步

2.如果获取锁失败，先执行**addWaiter(Node.*EXCLUSIVE*)** ，将当前正在执行的线程放入CLH等待队列，并标记为独占模式

3. **acquireQueued()**方法通过一个死循环，让刚入队的线程在队列中不断获取锁，如果获取失败，就通过parkAndCheckInterrupt()方法阻塞线程，直到线程再次被唤醒，然后再次尝试获取锁，一直循环到成功获取锁为止

由于**acquireQueued()**的源码过于难以理解，我们这里只看一看tryAcquire()方法的源码即可

protected boolean tryAcquire(int arg) {

throw new UnsupportedOperationException();  
}

是不是很惊讶？什么都没有实现，只抛出了一个不支持异常，可以去看一下tryAcquireShared()方法，也是只抛了个异常。这是因为AQS只是一个框架，虽然acquire()方法和acquireShared()等方法已经写好了，但这些方法所依赖的tryAcquire()和tryAcquireShared()方法需要各个子类去自行实现

那为什么不用abstract关键字来声明这些方法呢？这是因为独占模式下只用实现tryAcquire-tryRelease，而共享模式下只用实现tryAcquireShared-tryReleaseShared。如果都定义成abstract，那么每个模式也要去实现另一模式下的接口，不仅不规范，还增加了工作量。

##### 6.7.2.4.2释放锁

AQS通过release()方法释放独占锁，彻底释放了锁之后（即state=0），它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock()，下面我们来看看release()的源码

*/\*\**

*\* 独占模式下的释放操作，如果tryRelease()方法返回true，则通过解除一个或多个线程的阻塞来实现，此方法可用于实*

*\* 现Lock接口的unlock()方法。*

*\*/*

public final boolean release(int arg) {  
 //尝试释放独占锁，tryRelease由独占类型的子类实现

if (tryRelease(arg)) {  
 //如果释放成功

Node h = head;  
 //如果头节点不为空并且头节点的等待状态不为0就唤醒其后继节点

if (h != null && h.waitStatus != 0)  
 //唤醒该节点

unparkSuccessor(h);  
 return true;  
 }  
 return false;  
}

#### 6.7.2.5 共享模式（SHARED）

##### 6.7.2.5.1获取锁

AQS通过acquireShared()方法获取共享锁，我们来看一下它的源码

*/\*\**

*\*获取共享锁模式，中途忽略中断。通过调用至少一次tryAcquireShared ()方法实现，如果获取锁成功则返回。否则，线程*

*\*将排队。之后可能会出现反复阻塞(多次获取失败)和取消阻塞(再次尝试获取)的情况，调用tryAcquireShared()方法直*

*\*到成功。*

*\*/*public final void acquireShared(int arg) {  
 // tryAcquireShared由共享类型的子类实现

返回值小于0表示获取锁失败，需要进入等待队列。

返回值等于0表示当前线程获取共享锁成功，但它后续的线程是无法继续获取的，也就是不需要把它后面等待的节点唤醒。

如果返回值大于0，表示当前线程获取共享锁成功且它后续等待的节点也有可能继续获取共享锁成功，也就是说此时需要把后续节点唤醒让它们去尝试获取共享锁。

//尝试获取共享锁，返回值小于0表示获取失败

if (tryAcquireShared(arg) < 0)  
 //加入等待队列，然后循环尝试获取锁，每次尝试失败后都会挂起自己，等待下次尝试

doAcquireShared(arg);  
}

##### 6.7.2.5.2释放锁

*/\*\**

*\*独占模式下的释放操作，如果tryRelease()方法返回true，则通过解除一个或多个线程的阻塞来实现*

*\*/*

public final boolean releaseShared(int arg) {  
 //尝试释放共享锁，tryReleaseShared由共享类型的子类实现

if (tryReleaseShared(arg)) {  
 doReleaseShared();  
 return true;  
 }  
 return false;  
}

* + 1. Lock使用

Lock是jdk提供的一个接口，请注意：

1. **Lock的实例一般定义为成员变量**，如果定义为局部变量，每个线程都会保存一个自己的副本，那么在获取锁的操作时，实际每个线程获取的是不同的锁，无法形成同步互斥访问。
2. **获取锁的操作要放在try模块之外**，获取锁后的操作应当放在trycatch内，原因是如果获取锁放在try模块内的话，当获取锁的操作发生异常会调用finally中的代码释放锁，而此时并没有获取锁，就会抛出异常，之后的操作放在trycatch内，是防止获取锁后正常执行时发生异常，程序无法执行到释放锁的代码。

public interface Lock {

void lock();  
 void lockInterruptibly() throws InterruptedException; // 可以响应中断  
 boolean tryLock();  
 boolean tryLock(long time, TimeUnit unit) throws InterruptedException; // 可以响应中断  
 void unlock();  
 Condition newCondition();// 新建一个和自己关联的Condition  
}

下面来依次介绍这几个方法：

**1、void lock()**

lock()方法是平常使用得最多的一个方法，就是用来获取锁。如果锁已被其他线程获取，则进行等待。调用lock()方法后会处于无条件地轮询获取锁的等待状态中，并且**等待过程中不会响应中断**，除非成功获取到锁。

class ThreadLocalTest {

private Lock lock = new ......; //创建一种锁  
  
 public void getLock() {  
 lock.lock(); //获得锁  
 try {  
  *//TODO 同步任务处理*  
 } catch (Exception e) {

*//TODO 捕获异常*  
 } finally {  
 lock.unlock(); //释放锁  
 }  
 }  
}

**2、boolean tryLock()**

带有boolean型返回值，尝试获取锁，无论是否成功会立即返回，不进行等待，成功获取锁返回true，反之返回false。

class ThreadLocalTest {

private Lock lock = new ......; //创建锁  
  
 public void getLock() {  
  
 if(lock.tryLock()) {  
 try {  
 //*TODO 任务处理* } catch (Exception e) {  
 //*TODO 捕获异常* } finally {  
 lock.unlock(); //释放锁  
 }  
 } else {  
 //*TODO 获取锁失败，进行相应处理*  
 }  
 }  
}

**3、boolean tryLock(long time, TimeUnit unit) throws InterruptedException()**

尝试获取锁，如果获取不到，不立刻返回，而是等待**time时间，时间单位是unit，超时后还未获取到锁，直接返回**，如tryLock(4, TimeUnit.SECONDS)等待4秒，之后如果还未获取到锁，直接返回false；**可以在等待过程中响应中断**

**4、void lockInterruptibly() throws InterruptedException()**

lockInterruptibly是响应中断的lock()方法，首先尝试获取锁，获取不到就等待并不断重试，等待过程中响应中断

* + 1. 可重入的互斥锁ReentrantLock

ReentrantLock，意思是“可重入锁”，它是Lock接口的一个实现类，除实现Lock接口的方法外，还提供了非常丰富的其他的锁操作方法，如公平锁和非公平锁

* + - 1. 重入锁

可重入锁是指任意线程在获取到锁之后，再次获取该锁而不会被该锁所阻塞。这需要重入锁识别当前尝试获取锁的线程是否为此时占据锁的线程，并且要求线程重复n次获取锁后，必须n次释放锁，其他线程才能够获取该锁。重入意味着锁操作的颗粒度为“线程”。

重入锁的每个锁都**关联一个线程持有者和线程计数器**，当计数器为0时表示该锁没有被任何线程持有，那么任何线程都可能获得该锁而调用相应的方法；当某一线程请求成功后，JVM会记下锁的持有线程，并且将计数器置为1；此时其它线程请求该锁，则必须等待；而当前已持有该锁的线程如果再次请求同一个锁，就可以再次拿到这个锁，同时计数器会递增；当线程退出同步代码块时，计数器会递减，如果计数器为0，则释放该锁

可重入锁最大的意义在于可以防止死锁(后面会提到如何通过它预防死锁)，synchronized关键字其实也是一种可重入锁，比如一个类中的两个非静态方法都被synchronized修饰，则线程在获取synchronized锁访问一个同步方法时是可以进入另一个synchronized方法的(因为线程持有其对象/类锁)

public class Widget{

public synchronized void doSomething(){  
 //*TODO 父类方法执行任务*  
 }  
}  
  
public class LoggingWidget extends Widget{  
 public synchronized void doSomething(){  
 super.doSomething();  
 }  
}

//*TODO 假设这是一个线程*

Widget w = new LoggingWidget();

w.doSomething();

比如上面的例子，线程在执行时，只生成了LoggingWidget对象，虽然这里是利用多态的特性通过父类引用来调用的，但实际只生成了一个子类对象，并没有生成父类对象。线程在调用w.doSomething()时，首先获取子类LoggingWidget的对象锁，然后执行super.doSomething()， 此时再次尝试获取子类对象LoggingWidget的对象锁，如果synchronized是不可重入的，那么将不允许重复获取LoggingWidget的锁，线程就会一直阻塞，一直尝试获取一个无法获取的锁。super调用父类的方法实际上还是在子类中调用的，在JVM里面同时保存了父类和子类的被重写方法的实现，也就是说子类中有两个同名方法。

* + - 1. 公平锁和非公平锁

在多线程开发中，往往会出现一个线程对某个加锁资源长时间占有的情况，也就是我们常说的线程执行比较耗时。这时，在线程1占有锁的时候，往往会有线程2、3、4…在队列等待线程1释放锁资源，这些线程按先来后到的顺序进行排队，等待线程1释放锁资源。

**公平锁**：加锁前先查看是否有排队等待的线程，有的话优先处理排在前面的线程，先来先得。

**非公平锁**：线程加锁时直接尝试获取锁，获取不到就到线程等待队列的队尾等待。

ReentrantLock提供了一个抽象静态内部类Sync用作实现可重入锁的锁机制(Sync实现了AQS接口，使用了AQS的独占模式)，ReentrantLock还提供了两个继承Sync静态抽象内部类的内部子类，分别是静态内部类NonfairSync(非公平锁类)和静态内部类FairSync(公平锁类)，实现公平和非公平锁的类都会在锁内部维护的一个双向链表(这是由AQS提供的)，其Node结点的值就是每一个请求当前锁的线程。

**非公平锁性能比公平锁高5~10倍**

ReenTrantLock类在生成其实例对象时就要求指定锁的类型(公平or非公平，默认非公平)

**ReenTrantLock默认使用非公平锁**

public ReentrantLock() {

sync = new NonfairSync();  
}  
public ReentrantLock(boolean fair) {  
 sync = fair ? new FairSync() : new NonfairSync();  
}

ReenTrantLock加锁方法lock方法实现如下：

public class ReentrantLock implements Lock,java.io.Serializable {

private final Sync sync;

public void lock() {  
 sync.lock();  
 }

……

}

我们可以看到lock方法内部调用了sync的lock方法，在ReenTrantLock类的内部，还有两个内部类——FairSync和NonfairSync，它们是ReenTrantLock实现公平锁和非公平锁的关键

##### 6.7.4.2.1公平锁（Fair）

下面的源码我们可以看到，公平sync的lock()方法实际是调用了acquire()方法，acquire(1)代表获取锁的次数为1(如果已持有，就是重入次数为1)。在6.7.2中我们了解过，在AQS中acquire方法实际是调用了tryAcquire，而公平锁和非公平锁的父类Sync实现了AQS接口，源码如下

static final class FairSync extends Sync {

private static final long *serialVersionUID* = -3000897897090466540L;  
  
 final void lock() {  
 acquire(1);  
 }  
protected final boolean tryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 //在可重入锁ReentrantLock中，state代表锁当前已被重入的计数

int c = getState();  
 if (c == 0) {

//如果当前重入计数为0，即当前锁资源已释放s

//hasQueuedPredecessors翻译过来是“队列中有前驱节点”, 在hasQueuedPredecessors方法的注释中我们看到这样一句话——“此方法的调用相当于getFirstQueuedThread() != Thread.currentThread() && hasQueuedThreads(), 但本方法可能比它更高效。 如果在当前线程之前有排队的线程, 该方法将返回true, 如果当前线程在队列的头或者队列为空则返回false” 。hasQueuedPredecessors执行完毕后，CAS设置state状态为acquires  
 if (!hasQueuedPredecessors() && compareAndSetState(0, acquires)) {  
 //设置当前线程拥有锁资源的独占访问权

setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 else if (current == getExclusiveOwnerThread()) {  
 //当前试图获取锁的线程是正在使用锁的线程，即发生锁重入，acquires是本次重入增加的重入计数

int nextc = c + acquires;  
 if (nextc < 0)  
 //参数传入不正确导致nextc小于0，抛出异常

throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }

//tryAcquire失败后，AQS中的acquire方法会将线程加入到等待队列中  
 return false;  
 }  
}

如果ReenTrantLock希望使用公平锁，那么在实例化时应该这么做：

ReentrantLock lock = new ReentrantLock(true);

从源码中我们可以看出，在锁已被释放的情况下，!hasQueuedPredecessors()会使得只有出现以下两种情况，公平锁才会尝试修改state状态并设置当前线程持有锁资源

1. 当等待队列为空(即不需排队)
2. 当前线程就是队列的第一位(即排队已经轮到该线程了)

##### 6.7.4.2.2非公平锁（Nonfair）

ReentrantLock 实现的lock()方法默认采用的是非公平锁，相比于公平锁模式，非公锁的效率要快上很多

static final class NonfairSync extends Sync {

private static final long *serialVersionUID* = 7316153563782823691L;  
final void lock() {  
 //直接尝试插队获取锁资源，CAS尝试设置state的值，期望为0(期望锁没被占用)，尝试修改为1，如果锁被占用，则插队失败，否则设置当前线程拥有锁资源的独占访问权

if (compareAndSetState(0, 1))  
 setExclusiveOwnerThread(Thread.*currentThread*());  
 else  
 //插队失败，转入正常流程

acquire(1);  
 }  
  
 protected final boolean tryAcquire(int acquires) {  
 //调用父类Sync提供的nonfairTryAcquire方法

return nonfairTryAcquire(acquires);  
 }  
}

//为了方便理解，贴出父类Sync的nonfairTryAcquire方法

final boolean nonfairTryAcquire(int acquires) {  
 final Thread current = Thread.*currentThread*();  
 int c = getState();  
 if (c == 0) {  
 //以上步骤和公平锁一样，下面的判断体现了非公平所的“插队”思想

//如果锁资源已被释放，不用管自己是否排队，直接再次尝试获取锁资源，期望值为0，尝试修改为acquires，如果修改成功，则直接返回true

if (compareAndSetState(0, acquires)) {  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }

//当前试图获取锁的线程是正在使用锁的线程，即发生锁重入，acquires是本次重入增加的重入计数  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0) // overflow  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 //如果非公平锁获取锁失败，那么当前线程也将和在公平锁中的一样，进入CLH等待队列中

return false;  
}

从上述源码可以看出，公平锁和非公平锁加锁过程中最大的区别在于——非公平锁更强调”插队”，非公锁不管锁资源的状态，也不管队列中是否有线程排队，每次都是直接尝试获取锁，如果设置锁state状态失败，说明锁有人在使用。此时线程将进入等待队列，获取失败后的处理过程等同于公平锁，因为这个获取失败后的处理过程都在AQS独占模式的acquire方法中实现了

##### 6.7.4.2.3公平锁和非公平锁的释放tryRelease

由于公平锁和非公平锁释放的过程一样，因此tryRelease方法直接在二者的父类Sync中实现了

protected final boolean tryRelease(int releases) {

//获取当前重入次数并减去本次要释放的次数

int c = getState() - releases;  
 //如果当前线程不是正在使用锁的线程，抛出异常

if (Thread.*currentThread*() != getExclusiveOwnerThread())  
 throw new IllegalMonitorStateException();  
 boolean free = false;  
 //如果重入计数为0，则释放锁资源，否则只是减少重入计数而不释放锁

if (c == 0) {  
 free = true;  
 setExclusiveOwnerThread(null);  
 }

//设置重入计数值  
 setState(c);  
 return free;  
}

非公平锁之所以会比公平锁快这么多，是因为公平锁维护队列的开销比非公平锁大。对于公平锁来说，后来的线程想要加锁的话，即便锁空闲，也要先检查有没有其他线程在排队，如果有，后来的线程自己要挂起，并加到队列后面。在这种情况下，相比于非公平锁，公平锁还要去唤醒队列最前面的线程。多了一次挂起和唤醒的操作，性能自然有所损耗

* + - 1. ReentrantLock的优点

在JDK1.5中，ReentrantLock相比于synchronized性能更强，但自从jdk1.6对java内置锁进行优化后，ReentrantLock便不再占有性能上的优势，但ReentrantLock仍拥有一些内置锁无法企及的特性

1. **ReentrantLock可限时：**

ReentrantLock实现了lock接口，lock接口中提供了tryLock(long timeout, TimeUnit unit)方法，该方法可以限制获取锁的时间，当尝试获取锁的动作超时后，将终止获取操作。

这一特性还可以用于解决死锁，当某个线程无法在指定时间内获取所有的锁资源，就释放已获得的锁，返回失败等待重试

1. **ReentrantLock可中断：**

synchronized一旦尝试获取锁就会一直等待直到获取到锁。而ReentrantLock是可以响应中断的，lock的lockInterruptibly允许重入锁响应中断

* + - 1. Condition条件
         1. 接口定义

虽然ReentranLock比内置锁的加锁功能更强大，但synchronzied有诸多如wait，notify等方法配合使用。所幸，JUC包为我们提供了一个Condition接口来配合lock接口使用，帮助我们达成以上功能 。下面是Condition接口的接口定义：

public interface Condition {

//当前线程进入等待状态，直到被通知（signal）或者被中断时，线程进入运行状态，类似于Object对象的wait方法

void await() throws InterruptedException;

//当前线程进入等待状态，直到被通知（signal），对中断不做响应

void awaitUninterruptibly();

//在接口await的基础上增加了超时响应，返回值表示当前剩余的时间，如果在nanosTimeout之前被唤醒，返回值 = nanosTimeout - 实际消耗的时间，返回值 <= 0表示超时；该方法只有在超时或者超时前被唤醒才会结束执行，返回剩余时间

long awaitNanos(long nanosTimeout) throws InterruptedException;

//同样是在接口await的基础上增加了超时响应，与接口awaitNanos不同的是：

1.可以自定义超时时间单位；

2.返回值返回true/false，在time之前被唤醒，返回true，超时返回false。

boolean await(long time, TimeUnit unit) throws InterruptedException;

//当前线程进入等待状态，这个状态将最多保持到某个指定的时间点，如果没有到指定时间被通知（signal）返回true，否则，到达指定时间，返回false

boolean awaitUntil(Date deadline) throws InterruptedException;

//随机唤醒一个在Condition上等待的线程，相当于notify方法

void signal();

//唤醒在Condition上等待的所有线程，相当于notifyAll方法

void signalAll();  
}

一个Condition和一个lock关联在一起，如果希望创建一个Condition，可以通过lock接口调用newCondition，如：

private Lock lock = new ReentrantLock();

private Condition condition = lock.newCondition();

* + - * 1. 源码分析

Condition是放在AQS中实现的，ConditionObject是其在AQS中的实现类，因为Condition相关操作都需要获取锁，所以作为AQS的内部类也比较合理，这样做也可以利用AQS的FIFO队列进行控制

public class ConditionObject implements Condition, java.io.Serializable {

private static final long *serialVersionUID* = 1173984872572414699L;  
 */\*\* First node of condition queue. \*/* private transient Node firstWaiter;  
 */\*\* Last node of condition queue. \*/* private transient Node lastWaiter;

……

}

await方法

public final void await() throws InterruptedException {

if (Thread.*interrupted*())  
 throw new InterruptedException();  
 //将当前线程添加到等待队列的队尾

Node node = addConditionWaiter();

// 释放当前线程的锁，并唤醒后继节点的线程，失败则将当前线程设置为取消状态  
 int savedState = fullyRelease(node);  
 int interruptMode = 0;

// 如果没在同步队列  
 while (!isOnSyncQueue(node)) {  
 //就让线程等待(阻塞线程)

LockSupport.*park*(this);

// 如果有中断那么退出循环  
 if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)  
 break;  
 }  
 if (acquireQueued(node, savedState) && interruptMode != *THROW\_IE*)  
 interruptMode = *REINTERRUPT*;  
 if (node.nextWaiter != null) // clean up if cancelled  
 unlinkCancelledWaiters();  
 if (interruptMode != 0)  
 reportInterruptAfterWait(interruptMode);  
}

* + - * 1. Condition使用

还是仿照6.3.2中生产者消费者的例子(将下例中的singalAll()换成singal()就会发生死锁)

public class VolatileClass {

public static void main(String[] args) {  
 List<Integer> cache = new ArrayList();  
 Lock lock = new ReentrantLock();  
 Condition condition=lock.newCondition();  
 Producers producers = new Producers(cache, lock, "生产者—1",condition);  
 Consumers consumer1 = new Consumers(cache, lock, "消费者-1",condition);  
 Consumers consumer2 = new Consumers(cache, lock, "消费者-2",condition);  
  
 new Thread(consumer1).start();  
 new Thread(consumer2).start();  
 new Thread(producers).start();  
 }  
  
}  
  
class Producers implements Runnable {  
 private Lock lock;  
 private String name;  
 private static Random *random* = new Random();  
 private List<Integer> cache;  
 private Condition condition;  
  
 public Producers(List<Integer> cache, Lock lock, String name,Condition condition) {  
 this.lock = lock;  
 this.cache = cache;  
 this.name = name;  
 this.condition = condition;  
 }  
  
 @Override  
 public void run() {  
 while (!Thread.*currentThread*().isInterrupted()) {  
 this.produce();  
 }  
 }  
  
 private void produce() {  
 lock.lock();  
 try {  
 while (!cache.isEmpty()) {  
 System.*out*.println(this.name + ":进入等待池");  
 condition.await();  
 }  
 cache.add(*random*.nextInt());  
 System.*out*.println(this.name + ":生产一个整数");  
 condition.signalAll();  
 System.*out*.println(this.name + ":唤醒等待池中所有线程");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
}  
  
class Consumers implements Runnable {  
 private Lock lock;  
 private String name;  
 private List<Integer> cache;  
 private Condition condition;  
  
 public Consumers(List<Integer> cache, Lock lock, String name,Condition condition) {  
 this.cache = cache;  
 this.name = name;  
 this.lock = lock;  
 this.condition = condition;  
 }  
  
 @Override  
 public void run() {  
 while (true) {  
 this.consumer();  
 }  
 }  
  
 private void consumer() {  
 if (lock.tryLock()) {  
 while (cache.isEmpty()) {  
 System.*out*.println("缓存队列为空，消费者" + this.name + "等待");  
 try {  
 condition.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println(this.name + ":消费一个数据:" + cache.remove(0));  
 condition.signalAll();  
 System.*out*.println(this.name + ":唤醒等待池中所有线程");  
 lock.unlock();  
 }  
 }  
}

### 6.7.6 信号量(基于许可的多线程控制)

#### 6.7.6.1互斥和同步

互斥量用于线程的互斥，信号量则用于线程的同步。

**互斥：**是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

**同步：**是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源

**互斥量用于线程的互斥，而信号量用于线程的同步**。互斥量只能为0或1，代表着是否允许访问；而信号量可以为任何非负整数，代表某种资源剩余可访问次数，如AQS中的State属性。

#### 6.7.6.2 信号量Semaphore

Semaphore是计数信号量。Semaphore管理着一组资源访问许可(permit)。在每个线程尝试访问资源前，首先需要通过acquire方法获取许可(permit)。如果没有许可，acquire方法将会阻塞直到别的线程释放(release)了一个许可(或者发生中断，或者操作超时)

最常见的一个例子——数据库连接池。我们可以构造一个指定最大长度的连接池，当池为空时，请求资源的操作将会阻塞，直到 池非空(或者操作中断、超时)

Semaphore将AQS的state变量用于保存permit数量，同样的，信号量也存在公平信号量和非公平信号量。**公平信号量是指获得许可(permit)的顺序与线程所在的CLH队列中的顺序有关，非公平信息量就是无关的了。**

public Semaphore(int permits) {

sync = new NonfairSync(permits);  
}

public Semaphore(int permits, boolean fair) {  
 sync = fair ? new FairSync(permits) : new NonfairSync(permits);

}

#### 6.7.6.3 公平信号量(Fair)

线程在尝试获取信号量许可时，对于公平信号量而言，如果当前线程不在CLH等待队列的头部，则排队等候

static final class FairSync extends Sync {

private static final long *serialVersionUID* = 2014338818796000944L;  
  
 FairSync(int permits) {  
 super(permits);  
 }

//如果返回值小于0，则代表获取失败  
 protected int tryAcquireShared(int acquires) {  
 for (;;) {

//hasQueuedPredecessors翻译过来是“队列中有前驱节点”, 在hasQueuedPredecessors方法的注释中我们看到这样一句话——“此方法的调用相当于getFirstQueuedThread() != Thread.currentThread() && hasQueuedThreads(), 但本方法可能比它更高效。如果在当前线程之前有排队的线程, 该方法将返回true, 如果当前线程在队列的头或者队列为空则返回false” 。  
 if (hasQueuedPredecessors())

//如果不在队头，获取失败  
 return -1;

//如果在队头，尝试获取许可  
 int available = getState();  
 int remaining = available - acquires;  
 //如果remaining剩余许可< 0，失败返回一个负数，然后逻辑短路

//如果remaining剩余许可>=0，通过修改许可数量，并返回剩余许可数

if (remaining < 0 ||  
 compareAndSetState(available, remaining))  
 return remaining;  
 }  
 }  
}

#### 6.7.6.4 非公平信号量(NonFair)

对于非公平信号量而言，无论当前线程是不是在CLH等待队列的头部，它都会直接获取信号量

static final class NonfairSync extends Sync {

private static final long *serialVersionUID* = -2694183684443567898L;  
  
 NonfairSync(int permits) {  
 super(permits);  
 }  
  
 protected int tryAcquireShared(int acquires) {

//不管CLH队列，直接获取许可  
 return nonfairTryAcquireShared(acquires);  
 }  
}

#### 6.7.6.5 Semaphore的使用

假设某厕所共有10个坑位可以使用

public class ResourceManage {

private final Semaphore semaphore ;  
 private boolean resourceArray[];  
 private final ReentrantLock lock;  
 public ResourceManage() {  
 this.resourceArray = new boolean[10];//存放厕所状态   
 this.semaphore = new Semaphore(10,true);//控制10个共享资源的使用，使用公平模式进行共享;公平模式的信号量，等待队列最前面的优先获得信号量   
 this.lock = new ReentrantLock(true);//公平模式的锁，先来的先选   
 for(int i=0 ;i<10; i++){  
 resourceArray[i] = true;//初始化为资源可用的情况   
 }  
 }  
 public void useResource(int userId){  
 semaphore.acquire();  
 try{  
 int id = getResourceId();//占到一个坑   
 System.*out*.print("userId:"+userId+"正在使用资源，资源id:"+id+"\n");  
 Thread.*sleep*(100);//蹲坑100ms   
 resourceArray[id] = true;//退出这个坑   
 }catch (InterruptedException e){  
 e.printStackTrace();  
 }finally {  
 semaphore.release();//释放信号量，计数器加1   
 }  
 }  
 //获取可用的坑位id，并占坑，对于单个坑位，应当使用排他锁独占坑位

private int getResourceId(){  
 int id = -1;  
 lock.lock();  
 try {  
 //虽然使用了锁控制同步，但由于只是简单的一个数组遍历，效率还是很高的。   
 //遍历坑位，寻找可蹲的坑

for(int i=0; i<10; i++){  
 if(resourceArray[i]){  
 resourceArray[i] = false;  
 id = i;  
 break;  
 }  
 }  
 }catch (Exception e){  
 e.printStackTrace();  
 }finally {  
 lock.unlock();  
 }  
 return id;  
 }  
}

## 6.9 死锁

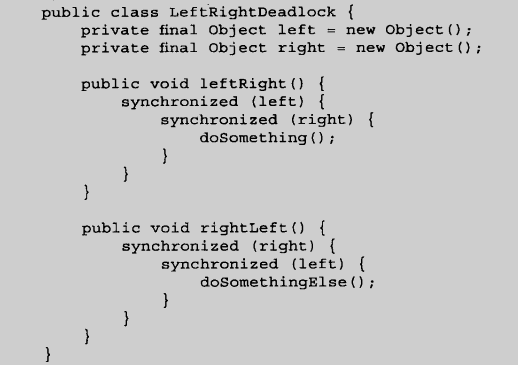
### 6.9.1 死锁的常见类型

线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，这些线程处于等待状态，互相等待对方释放资源，无法继续执行的现象

产生死锁需要满足如下四个条件：

（1） 互斥条件：一个资源每次只能被一个线程使用，不能同时被两个或两个以上的线程占有。  
（2） 请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。  
（3） 不剥夺条件: 线程已获得的资源，在末使用完之前，不能强行剥夺。  
（4） 循环等待条件: 当发生死锁时，所等待的线程必定会形成一个头尾相接的环路。即线程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

#### 6.9.1.1 简单的锁顺序死锁



上述代码中，如果有两个以上的线程，分别执行leftRight()和rightLeft()，就容易发生死锁，这是请求锁的顺序不一致导致的。

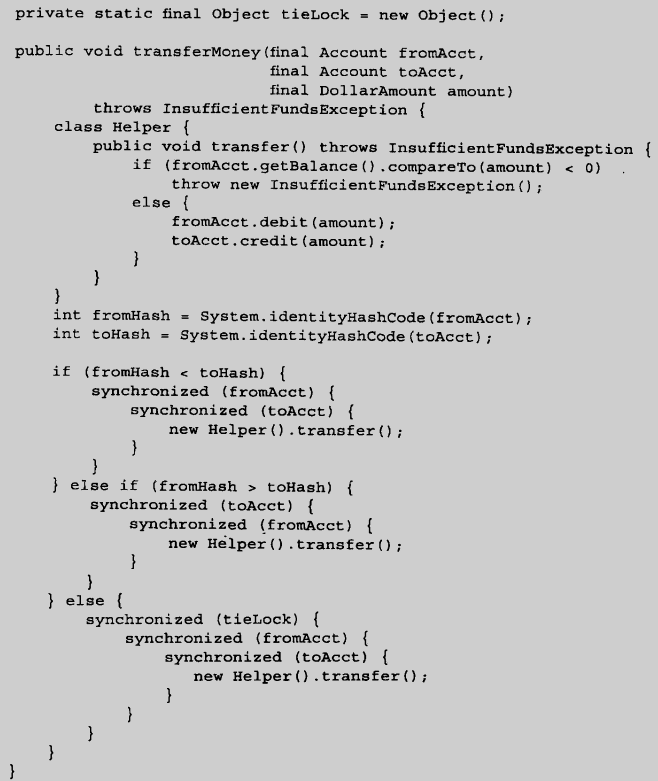
#### 6.9.1.2 动态的锁顺序死锁

如果锁资源是由参数传入的，那么情况就会复杂一些，比如下面的转账操作，需要锁住转账账号和被转账账号，看起来似乎不会发生死锁，所有的线程都按照先锁fromAccount，再锁toAccount的顺序执行



但可惜的是，上述代码会发生死锁，因为fromAccount和toAccount的顺序是由参数传入的顺序决定的，比如，X在给Y转账的同时(fromAccount是X，toAccount是Y)，Y也在给X转账(fromAccount是Y，toAccount是X)，这样一来，第一个线程先锁了X，再锁Y，第二个线程先锁Y，再锁X，就会发生锁顺序导致的死锁

解决这种死锁，还是要通过保证锁顺序来做到避免死锁，那么如何保证锁顺序呢？我们可以思考一下，参数顺序是可以改变的，但传入对象的地址是不变的，因此我们可以根据hash值的大小进行锁顺序控制。但是这样又有一个问题，在少数情况下，hash值也存在相等的可能，因此可以引入第三个锁，一旦发生hash碰撞，就强制串行执行程序。

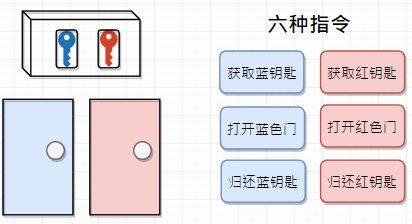


当然了，在实际业务中，账号一般存在一个id，这个id是从数据库中得出，我们完全可以根据账号id的大小来决定其执行顺序，而不必根据hash值决定

### 6.9.2 解决死锁的办法

要避免出现死锁的问题，只需要破坏四个条件中的任何一个就可以了。

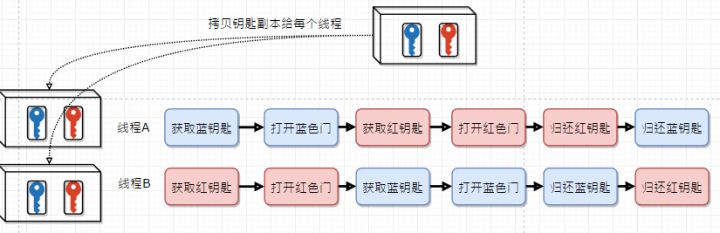
假设我们有一把蓝钥匙，可以打开一扇蓝色的门；以及一把红钥匙，可以打开一扇红色的门。两把钥匙被保存在一个皮箱里。同时我们定义六种行为：获取蓝钥匙，打开蓝色门，归还蓝钥匙，获取红钥匙，打开红色门，归还红钥匙。规则是，只有当红色门和蓝色门都被**同一个人(线程)**打开时，才可以进入门内



#### 6.9.2.1 破坏互斥条件(不实用)

**互斥条件：一个资源每次只能被一个线程使用，不能同时被两个或两个以上的线程占有**

只有一副钥匙，这是形成死锁的最关键的原因之一。显然，如果我们能在两个线程运行之前，能给每个线程单独拷贝一份钥匙的副本，就能有效的避免死锁了。



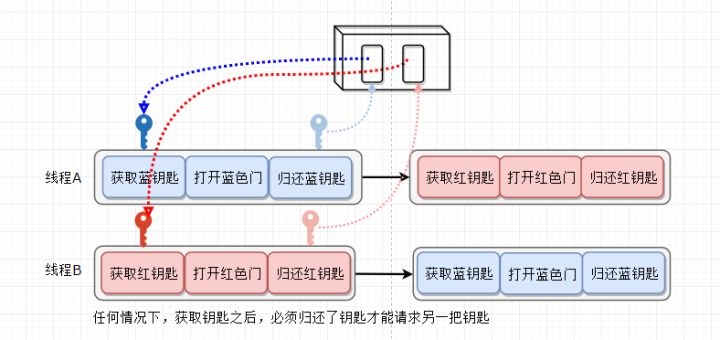
当然，这种方法使用限制极大，资源能否复用，常常并不是由我们决定的，比如打印机，即使可以复用，拷贝副本的代价常常是极大的，因此不推荐这种方法。

#### 6.9.2.2 破坏请求与保持条件

**请求与保持条件：一个线程因请求资源而阻塞时，对已获得的资源保持不放。**

任何一个线程“贪心”，都可能会导致死锁。大致意思就是说有了一把钥匙没还，就要另一把。

对于这种情况，我们可以采用资源预先分配策略，即线程在运行前一次申请完它所需要的全部资源。也就是说，对于任何一个线程，要么同时拿到两把钥匙，要么一把也不拿

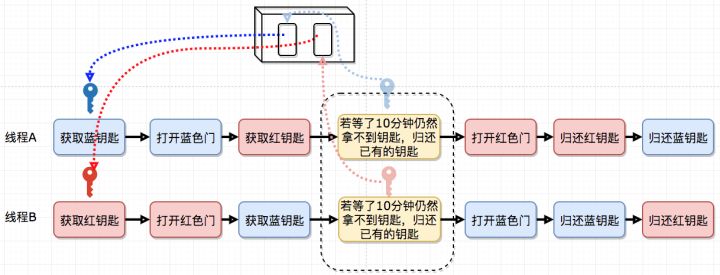


这种方式也有着极大的限制。首先，我们往往无法在线程执行之前得知它所需的所有资源；其次，该方法资源利用率低下，降低了并发性，即使有些资源最后才被该进程用到一次，但该进程在生存期间却一直占有它们，造成长期占着不用的状况

#### 6.9.2.3 破坏不剥夺条件

**不剥夺条件: 线程已获得的资源，在末使用完之前，不能强行剥夺。**

除非线程自己还钥匙，否则线程会一直占有钥匙，是形成不可剥夺条件的原因。这里，我们可以通过设置一个”最长占用时间“的阈值来解决这个问题——如果过了10分钟仍然没有进入下一个步骤，则归还已有的钥匙。这样的话，两个线程都能取到所需的钥匙继续下去了。



#### 6.9.2.4 破坏循环等待条件

**循环等待条件: 当发生死锁时，所等待的线程必定会形成一个头尾相接的环路。即线程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。**

发生死锁，要么是一个线程先取了红钥匙而另一个线程先取了蓝钥匙，要么是反过来，从而导致了可能形成了“环路等待”

所以我们可以强制规定任何线程取钥匙的顺序只能是 “先取蓝钥匙再取红钥匙”(或者反过来)，就能避免死锁了



#### 6.9.2.5 使用开放调用(open call)缩小锁粒度

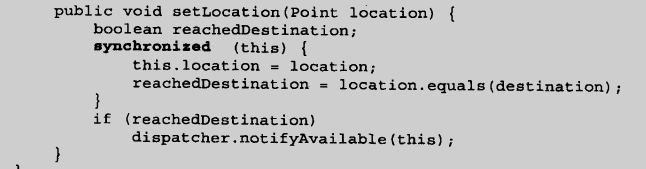
开放调用是指调用某个方法时不需要持有锁，也就是说synchronized不要直接加在方法上，尽量缩小锁的粒度，比如：

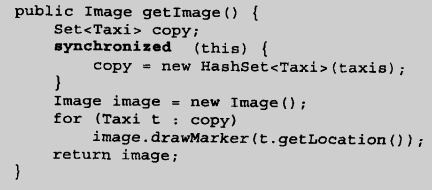
class Taxi {

private Point location, destination;//出租车当前位置和目的地  
 private final Dispatcher dispatcher;//所属车队  
  
 public Taxi(Dispatcher dispatcher) {  
 this.dispatcher = dispatcher;  
 }  
  
 // 获取出租车当前位置  
 public synchronized Point getLocation() {  
 return location;  
 }  
  
 // 设置位置,非开放调用,首先将获取当前对象(this)的对象锁  
 // 然后执行notifyAvailable方法时，尝试获取dispatcher对象的对象锁  
 public synchronized void setLocation(Point location) {  
 this.location = location;  
 if (location.equals(destination))  
 dispatcher.notifyAvailable(this);  
 }  
}  
  
class Dispatcher {  
 private final Set<Taxi> taxis;//所有taxi  
 private final Set<Taxi> availableTaxis;//当前可用taxi  
  
 public Dispatcher() {  
 taxis = new HashSet<Taxi>();  
 availableTaxis = new HashSet<Taxi>();  
 }  
 //将taxi加入到可用set集合中去  
 public synchronized void notifyAvailable(Taxi taxi) {  
 availableTaxis.add(taxi);  
 }  
 //在地图上画出所有出租车的点，首先获取dispatcher对象(this)的对象锁  
 //然后getLocation方法将会依次获取所有taxi对象的对象锁  
 public synchronized Image getImage() {  
 Image image = new Image();  
 for (Taxi t : copy)  
 image.drawMarker(t.getLocation());  
 return image;  
 }  
}

setLocation方法和getImage方法获取锁的顺序相反，可能发生死锁

上述代码可以修改为：





可以看到修改后的代码缩小了锁的粒度，两个锁不再重叠，在其中一方请求第二个锁资源前，会释放当前所持有的锁资源(该锁资源已使用完毕)，避免了死锁的发生。

但是该方法有一个缺点，就是可能导致原本的原子性操作变为非原子性。但是在上例中并不需要担心这个问题，更新taxi当前的位置 和 将该出租车重新加入可用set中这两个操作并不需要合并为一个原子操作，同理getImage也不需要作为一个原子方法。

## 6.10 线程池

在之前，每当我们需要一个线程的时候，都由我们自己手动去创建一个线程。这种使用方式在线程数量少的时候没什么问题，但当线程数量达到一定数量级并且每个线程都是执行一个时间很短的任务就结束了的时候，这样频繁地创建和销毁线程，会大大降低多线程系统的吞吐量。为了达到使线程执行完一个任务后不被销毁，能够继续执行其他的任务的目的，java提供了线程池技术

### 6.10.1 ThreadPoolExecutor类

#### 6.10.1.1 构造函数

从jdk1.5起，java提供了一个线程池实现类ThreadPoolExecutor。首先我们来看看他的构造函数：

public ThreadPoolExecutor(int corePoolSize, //核心线程池大小

int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler) {  
 if (corePoolSize < 0 ||  
 maximumPoolSize <= 0 ||  
 maximumPoolSize < corePoolSize ||  
 keepAliveTime < 0)  
 throw new IllegalArgumentException();  
 if (workQueue == null || threadFactory == null || handler == null)  
 throw new NullPointerException();  
 this.corePoolSize = corePoolSize;  
 this.maximumPoolSize = maximumPoolSize;  
 this.workQueue = workQueue;  
 this.keepAliveTime = unit.toNanos(keepAliveTime);  
 this.threadFactory = threadFactory;  
 this.handler = handler;  
}

**corePoolSize：**核心线程池大小。在默认情况下，刚刚创建好的线程池中不会有任何线程，如果有任务来了就创建一个线程(不论此时是否有空闲线程)，直到线程池中的线程数目达到corePoolSize，之后再有任务就将加入到缓存队列workQueue中。在线程数达到corePoolSize之前创建的是核心线程，核心线程即使长期闲置也依然会存在，超过corePoolSize之后就是非核心线程，非核心线程线程到一定时间会被销毁。当然，也可以通过prestartAllCoreThreads()或者prestartCoreThread()预创建corePoolSize个或一个线程

**maximumPoolSize：**线程池最大线程数，它表示在线程池中最多能创建多少个线程，线程池中的线程数(核心线程+非核心线程)永远不会超过这个数量

**keepAliveTime：**表示线程没有任务执行时最多保持多久时间会终止。即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。

**unit：**参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.*DAYS*; //天

TimeUnit.*HOURS*; //小时  
TimeUnit.*MINUTES*; //分钟  
TimeUnit.*SECONDS*; //秒  
TimeUnit.*MILLISECONDS*; //毫秒  
TimeUnit.*MICROSECONDS*; //微妙  
TimeUnit.*NANOSECONDS*; //纳秒

**workQueue：**一个阻塞队列，用来存储等待执行的任务，其类型将影响线程池的运行方式。当所有的核心线程都在干活时，新添加的任务会被添加到这个队列中等待处理，如果workQueue队列满了，则新建非核心线程执行任务。一般有如下三种选择：

**(1)ArrayBlockingQueue：**可以限定队列的长度，接收到任务的时候，如果没有达到corePoolSize的值，则新建线程(核心

线程)执行任务，如果达到了，则入队等候，如果队列已满，则新建线程(非核心线程)执行任务。再如果ArrayBlockingQueue队列满了，总线程数也达到了maximumPoolSize，则发生错误

**(2)LinkedBlockingQueue（常用）：** 这个队列接收到任务的时候，如果当前线程数小于核心线程数，则新建线程(核心线程)处理任务；如果当前线程数等于核心线程数，则进入队列等待。由于这个队列没有最大值限制，即所有超过核心线程数的任务都将被添加到队列中，这也就导致了maximumPoolSize的设定失效，因为总线程数永远不会超过corePoolSize

**(3)SynchronousQueue：** 这个队列接收到任务的时候，会直接提交给线程处理，而不保留它，如果所有线程都在工作怎么办？那就新建一个线程来处理这个任务！所以为了保证不出现 “rejectedExecution(线程数达到了maximumPoolSize而不能新建线程)” 的错误，使用这个类型队列的时候，maximumPoolSize一般指定成Integer.MAX\_VALUE，即无限大

**handler：**当拒绝处理任务时所要执行的策略

**(1)ThreadPoolExecutor.AbortPolicy(默认):** 直接丢弃任务并抛出RejectedExecutionException异常，jdk默认策略，

如果使用该策略，要用日志记录哪些任务被丢弃了。

**(2)ThreadPoolExecutor.DiscardPolicy：**也是丢弃任务，只是不抛出异常。  
**(3)ThreadPoolExecutor.DiscardOldestPolicy：**丢弃队列最前面的任务，然后重新尝试执行任务（如果再次失败则重复此过程）  
**(4)ThreadPoolExecutor.CallerRunsPolicy：**由调用线程处理该任务，就是直接由调用execute方法的线程本身来执行，但是很有可能造成当前线程也被阻塞

#### 6.10.1.2 线程池的执行流程

当线程池接收到一个任务时，如果工作线程数没有达到corePoolSize，那么就会新建一个线程，并绑定该任务，直到工作线程的数量达到 corePoolSize 前都不会重用之前的线程。

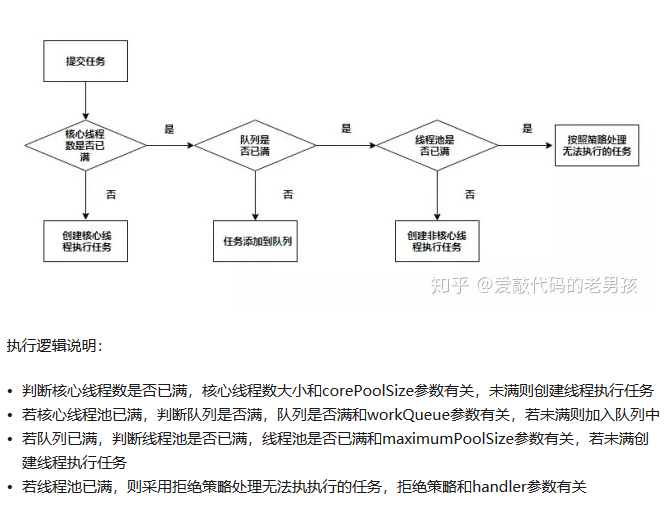
当工作线程数达到 corePoolSize 了，这时又接收到新任务时，会将任务存放在一个阻塞队列中等待核心线程去执行。为什么不直接创建更多的线程来执行新任务呢，原因是核心线程中很可能已经有线程执行完自己的任务了，或者有其他线程马上就能处理完当前的任务，并且接下来就能投入到新的任务中去，所以阻塞队列是一种缓冲的机制，给核心线程一个机会让他们充分发挥自己的能力。另外一个值得考虑的原因是，创建线程毕竟是比较昂贵的，不可能一有任务要执行就去创建一个新的线程。

所以我们需要为线程池配备一个阻塞队列，用来临时缓存任务，这些任务将等待工作线程来执行。接下来分为两种情况，一种是有界队列，一种是无界队列

如果是无界队列，那么当核心线程都在忙的时候，所有新提交的任务都会被存放在该无界队列中，这时最大线程数将变得没有意义，因为阻塞队列不会存在任务被装满的情况，除非超过int最大值(不过可能内存会先爆掉)

如果是有界队列，那么当阻塞队列中装满了等待执行的任务，这时再有新任务提交时，线程池就需要创建新的“临时”线程来处理，相当于增派人手来处理任务

如果新建线程超过maximumPoolSize仍处理不及，则触发拒绝策略



#### 6.10.1.3 线程池状态

ctl是对线程池的运行状态和线程池中有效线程的数量进行控制的一个字段， 它包含两部分的信息: 线程池的运行状态 (runState) 和线程池内有效线程的数量 (workerCount)

private final AtomicInteger ctl = new AtomicInteger(*ctlOf*(*RUNNING*, 0));

private static final int *COUNT\_BITS* = Integer.*SIZE* - 3;  
private static final int *CAPACITY* = (1 << *COUNT\_BITS*) - 1;  
  
// runState is stored in the high-order bits  
private static final int *RUNNING* = -1 << *COUNT\_BITS*;  
private static final int *SHUTDOWN* = 0 << *COUNT\_BITS*;  
private static final int *STOP* = 1 << *COUNT\_BITS*;  
private static final int *TIDYING* = 2 << *COUNT\_BITS*;  
private static final int *TERMINATED* = 3 << *COUNT\_BITS*;

通过上述源码我们可以看到，ctl使用了Integer类型来保存，高3位保存runState，低29位保存workerCount。

1.COUNT\_BITS 就是29，CAPACITY就是1左移29位减1（29个1），CAPACITY表示workerCount的上限值，大约是5亿。

2.高三位的runState有五种状态：

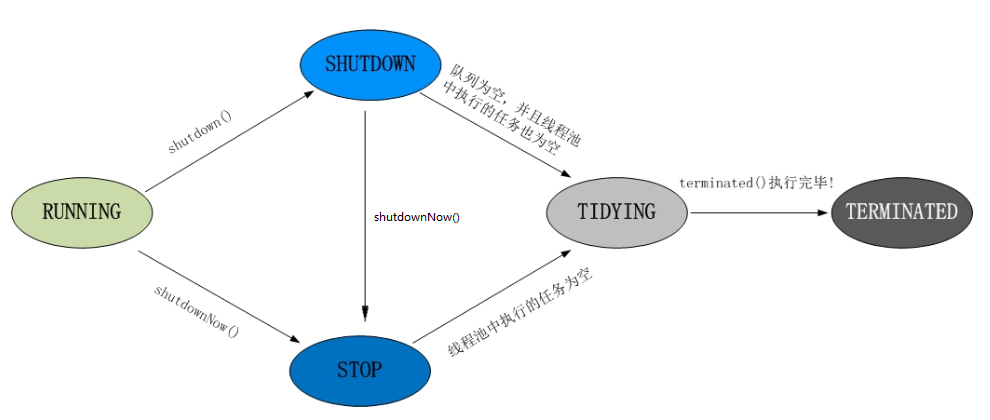
**·RUNNING（运行）：**线程池被一旦被创建，就处于RUNNING状态，此时线程池能接受新提交的任务，并且也能处理阻塞队列中的任务；

**·SHUTDOWN（关闭）：**关闭状态，不再接受新提交的任务，但却可以继续处理阻塞队列中已存在的任务。在线程池处于 RUNNING 状态时，调用 shutdown()方法会使线程池进入到该状态。（finalize() 方法在执行过程中也会调用shutdown()方法进入该状态）；

**·STOP（停止）：**不再接受新任务，也不处理队列中的任务，并且还会中断正在处理任务的线程。在线程池处于 RUNNING 或 SHUTDOWN 状态时，调用 shutdownNow() 方法会使线程池进入到该状态；

**·TIDYING（整理）：**如果所有的任务都已终止了(SHUTDOWN处理完剩余任务，STOP停止所有任务)，此时workerCount (有效线程数) 为0，线程池进入该状态，然后会调用 terminated() 方法进入TERMINATED 状态。terminated()是个钩子方法(方法体为空的方法)，若想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。

**·TERMINATED（终止）：**在terminated() 方法执行完后进入该状态(terminated方法是自动执行的，不需要手动调用)，此时线程池彻底终止。



#### 6.10.1.4 常用方法

**1.shutdown()：**

使当前未执行的线程继续执行，而不再添加新的任务Task，并使线程池进入SHUTDOWN状态

**2. shutdownNow()**

不再添加新任务，中断所有任务task，并且抛出InterruptedException异常，而未执行的线程不再执行，从任务队列中清除，并使线程池进入STOP状态。

该方法将s返回一个llist<Runnable>队列来存储未运行的任务

**3. isShutdown()**

判断线程池是否处于SHUTDOWN状态

**4. isTerminating()**

在调用shutdown()或shutdownNow()后，线程池开始进行终止前的准备(如停止当前执行中的任务、调用terminated()方法等)，但在线程池完全终止之前，该方法返回true

**5. isTerminated ()**

判断线程池是否已经完全终止

**6. awaitTermination (long timeout, TimeUnit unit)**

如果池中有任务正在被执行，将阻塞当前调用awaitTermination方法的线程，直到线程池处于TERMINATED状态 或 timeout等待超时

**7. prestartCoreThread() / prestartAllCoreThreads()**

前者每次调用一次就创建一个核心线程，返回的是boolean

后者启动全部核心线程，返回的是启动核心线程的数量

**8. remove(Runnable task)**

可以删除尚未被执行的Runnable任务。

**9. 多个set方法**

**setCorePoolSize：**设置核心池大小

**setMaximumPoolSize：**设置线程池最大能创建的线程数目大小

**10. 多个get方法**

**getActiveCount() :** 获取当前有多少正在执行任务的线程

**getPoolSize() :** 获取当前池中线程总数，包括正在执行任务的线程，也包括闲置的线程

**getCompletedTaskCount() ：**获取已经执行完成的任务数

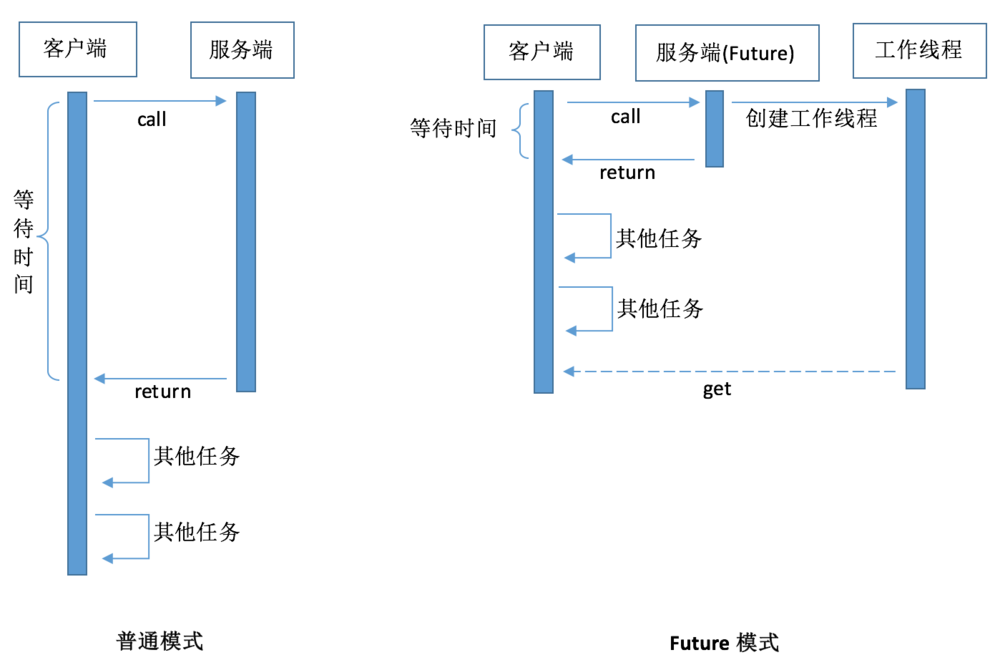
**getCorePoolSize() :** 获取线程池的corePoolSize值

**getMaximumPoolSize() :** 获取线程池的MaximumPoolSize值

**getTaskCount() ：**获取有多少个任务发送给了线程池

### 6.10.2 Futrue模式

java项目编程中，为了充分利用计算机CPU资源，一般开启多个线程来执行异步任务。但不管是继承Thread类还是实现Runnable接口，都无法获取任务执行的结果，如果需要获取执行结果，就必须通过共享变量或者使用线程通信的方式来达到效果，这样使用起来就比较麻烦。为了解决这个问题，JDK 5中引入了Callable和Future，通过它们执行异步任务可以获取执行结果。



#### 6.10.2.1 Callable、Future和RunnableFuture

public interface Callable<V> {

V call() throws Exception;  
}

Callable位于java.util.concurrent包下，它也是一个接口，在它里面只声明了一个叫做call()的泛型方法，该方法带有一个泛型返回值，并且可以抛出异常，我们可以将其当作有返回值的run()方法

Future接口可以用于查询异步执行的方法的结果，下面是Future接口的源码：

*/\*\**

*\* Future表示异步计算的结果。该接口提供了用于检查计算是否完成的方法和等待其完成并取回计算结果的方法等。*

*\* FutureTask是Future和Runnable接口的一个实现类，因此它可以被线程池执行*

*\*/*

public interface Future<V> {

*/\*\*  
 \* 试图取消此任务的执行。如果任务已经完成、已被取消或由于其他原因无法取消，则此操作将失败。  
 \* 1.如果调用此方法时任务还未开始，则任务将会被取消  
 \* 2.如果调用此方法时任务任务已启动，则*mayInterruptIfRunning*参数将确定任务是否应该中断以尝试取消任务  
 \** mayInterruptIfRunning*正在执行任务的线程是否应该中断*

*\*/* boolean cancel(boolean mayInterruptIfRunning);  
  
 */\*\*  
 \* 如果此任务在正常完成之前被取消，则返回 true  
 \*/* boolean isCancelled();  
  
 */\*\*  
 \* 如果任务完成，调用该方法将返回true，这里的完成包括 正常终止、异常 或 取消  
 \*/* boolean isDone();  
  
 */\*\*  
 \*等待计算完成(如果必要的话)，然后再取回其结果  
 \*/* V get() throws InterruptedException, ExecutionException;  
  
 */\*\*  
 \* 相比于上面的get方法，给定了一个最大等待时间  
 \*/* V get(long timeout, TimeUnit unit)  
 throws InterruptedException, ExecutionException, TimeoutException;  
}

为了能够成功接受到Runnable接口成功执行run()方法之后的结果，jdk提供了一个继承了Runnable和Future的接口——RunnableFuture，该接口的实现类是FutureTask，它实现了启动和取消线程，查询线程计算是否完成等功能，如果计算没有完成，get方法会阻塞，一旦计算完成，这个计算将不能被重启和取消

#### 6.10.2.2 FutureTask

如上文所述，FutureTask是Future接口的一个实现类，FutureTask能用来包装一个Callable或Runnable对象，因为它实现了Runnable接口，而且它能被传递到Executor进行执行。

FutureTask有几个重要的属性：

*/\*\* 用于存储FutureTask包装的Callable对象 \*/*

private Callable<V> callable;  
*/\*\* get()方法将要返回的 任务执行结果 或 异常 \*/*private Object outcome; // non-volatile, protected by state reads/writes

//任务的运行状态，初始为NEW，COMPETING和INTERRUPTING用的进行时语法，表示瞬时状态，存在时间极短，NORMAL代表

顺利完成；EXCEPTIONAL代表执行过程出现异常；CANCELED代表执行过程被取消；INTERRUPTED被中断

private volatile int state;

private static final int *NEW* = 0; //任务初始创建的状态  
private static final int *COMPLETING* = 1;//正在完成状态。在任务执行完毕后，结果被设置给outcome之前。该状态存在时间极短  
private static final int *NORMAL* = 2; //已完成状态，结果被设置给outcome后处于该状态  
private static final int *EXCEPTIONAL* = 3; //当任务在被执行的过程中抛了异常,FutureTask会将异常信息设置给FutureTask的outcome属性。在设置之前会将FutureTask的状态修改为COMPLETING,在设置之后会将FutureTask的状态修改为EXCEPTIONAL  
private static final int *CANCELLED* = 4; //当外部想要取消任务,而又不允许当任务正在执行的时候被中断时会将FutureTask的状态修改为CANCELLED。cancel(boolean mayInterruptIfRunning)的参数决定是否允许执行时中断，如果不允许，cancel仅设置状态为CANCELLED，然后立即返回true  
private static final int *INTERRUPTING* = 5; //当外部想要取消任务,同时允许当任务正在执行的时候被中断时,会先将FutureTask的状态设置为INTERRUPTING。瞬时状态，存在时间极短  
private static final int *INTERRUPTED* = 6; //当外部想要取消任务,同时允许当任务正在执行的时候被中断时,会先将FutureTask的状态设置为INTERRUPTING,然后设置执行任务的线程的中断标记位(线程中断),最后将Future的状态设置为INTERRUPTED

状态的变化只有以下4种可能：

**1.执行过程顺利完成：NEW -> COMPLETING -> NORMAL**

**2.执行过程出现异常：NEW -> COMPLETING -> EXCEPTIONAL**

**3.执行过程被取消：NEW -> CANCELLED**

**4.执行过程中，线程中断：NEW -> INTERRUPTING -> INTERRUPTED**

我们再来看一看其run方法：

public void run() {

//状态为NEW且无线程用于运行当前任务,否则返回

if (state != *NEW* ||!*UNSAFE*.compareAndSwapObject(this, *runnerOffset*, null, Thread.*currentThread*()))  
 return;  
 try {  
 Callable<V> c = callable;  
 if (c != null && state == *NEW*) {  
 V result;  
 boolean ran;  
 try {

//执行call()方法，并获取返回值  
 result = c.call();  
 //为true代表没有发生异常

ran = true;  
 } catch (Throwable ex) {  
 result = null;  
 ran = false;  
 //发生异常，将异常设置到outcome属性中

setException(ex);  
 }  
 if (ran)  
 //没有发生异常则将返回值设置到outcome中

set(result);  
 }  
 } finally {  
 // runner must be non-null until state is settled to  
 // prevent concurrent calls to run()  
 runner = null;  
 // state must be re-read after nulling runner to prevent  
 // leaked interrupts  
 int s = state;  
 if (s >= *INTERRUPTING*)  
 handlePossibleCancellationInterrupt(s);  
 }  
}

#### 6.10.2.3 使用示例

class Thread\_1 implements Callable<String> {

private int num = 1000;  
 @Override  
 public String call() throws Exception {  
 while (this.num-- > 0) {  
 System.*out*.println("num = " + num);  
 }  
 return String.*valueOf*(num);  
 }  
}

@Test

public void test\_1() {

Executor executor=Executors.*newSingleThreadExecutor*();  
 Callable<String> callable=new Thread\_1();  
 FutureTask<String> task = new FutureTask<>(callable);  
 executor.execute(task);  
 try {

// get()方法将会阻塞直到任务执行完毕，然后将结果返回  
 System.*out*.println("task.get() = " + task.get());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
}

void exec() throws InterruptedException, ExecutionException{  
 //进行异步任务列表  
 List<FutureTask<Integer>> futureTasks = new ArrayList<FutureTask<Integer>>();  
 //线程池 初始化十个线程 和JDBC连接池是一个意思 实现重用   
 ExecutorService executorService = Executors.newFixedThreadPool(10);  
 long start = System.*currentTimeMillis*();  
 //类似与run方法的实现 Callable是一个接口，在call中手写逻辑代码  
 Callable<Integer> callable = new Callable<Integer>() {  
 @Override  
 public Integer call() throws Exception {  
 Integer res = new Random().nextInt(100);  
 Thread.*sleep*(1000);  
 System.*out*.println("任务执行:获取到结果 :"+res);  
 return res;  
 }  
 };  
  
 for(int i=0;i<10;i++){  
 //创建一个异步任务  
 FutureTask<Integer> futureTask = new FutureTask<Integer>(callable);  
 futureTasks.add(futureTask);  
 //提交异步任务到线程池，让线程池管理任务 。  
 //由于是异步并行任务，所以这里并不会阻塞  
 executorService.submit(futureTask);  
 }  
  
 int count = 0;  
 for (FutureTask<Integer> futureTask : futureTasks) {  
 //futureTask.get() 得到我们想要的结果   
 //该方法有一个重载get(long timeout, TimeUnit unit) 第一个参数为最大等待时间，第二个为时间的单位  
 count+= futureTask.get();  
 }  
 long end = System.*currentTimeMillis*();  
 System.*out*.println("线程池的任务全部完成:结果为:"+count+"，main线程关闭，进行线程的清理");  
 System.*out*.println("使用时间："+(end-start)+"ms");  
 //清理线程池   
 executorService.shutdown();  
  
}

# 7.JAVA8 新特性

JAVA8是java提供长期支持的，一个具有划时代意义的版本，其提供了大量令人激动的特性，这些特性被广泛应用于日常编程中，下面我们将学习这些新特性

## 7.1 lambda表达式

Lambda 表达式，也可称为闭包，是一种函数式编程，它可以帮助我们使得代码变得更加简洁。Lambda允许程序员把函数作为参数传递进方法中。

这是lambda表达式的语法格式：

(parameters) -> expression

或  
(parameters) ->{ statements; }

Lambda表达式需要：

**可选的参数圆括号：**一个参数无需定义圆括号，但多个参数需要定义圆括号。eg：

(str) -> ""

(x,y) -> x - y

(int x, int y) -> x + y //接受两个int型参数，返回二者之和

**可选的大括号：**如果是一个具有返回值的表达式则不需要大括号；如果是多个语句则需要大括号，并且大括号内需要返回语句，eg：

str -> ""

str -> {return "";}

本质上，lambda表达式是把 接口的方法体 以lambda表达式 的形式表示出来：

interface MathOperation{

int ss(int a,int b);  
}  
MathOperation addition = (int a, int b) -> a + b;

## 7.2 方法引用

方法引用通过方法的名字来指向一个方法，帮助程序员简化代码。我们常常使用方法引用来简写lambda表达式中已经存在的方法

方法引用被用来直接访问类或者实例的已经存在的方法。方法引用是一个Lambda表达式，其使用的操作符是双冒号"::"。有时候，我们的Lambda表达式可能仅仅调用一个已存在的方法，而不做任何其它事，对于这种情况，通过一个方法名字来引用这个已存在的方法会更加清晰

有如下四种方式来使用方法引用：

|  |  |
| --- | --- |
| **引用静态方法** | **ContainingClass::staticMethodName** |
| **引用某个对象的实例方法** | **containingObject::instanceMethodName** |
| **引用某个类型的任意对象的实例方法** | **ContainingType::methodName** |
| **引用构造方法** | **ClassName::new** |

## 7.3 函数式接口

函数式接口是一种特殊的接口，相比于常见接口，函数式接口有如下特点：

1. 接口需要被@FunctionalInterface注解声明
2. 接口**有且只能有**一个抽象方法(没有方法体的方法声明)
3. 可以有多个非抽象方法(拥有方法体的方法)，但这些方法必须被default声明

Eg：

@FunctionalInterface

interface MyLambda{  
 int add(int a,int b);  
 default int reduce(int a,int b){return a-b;}  
}

//使用lambda表达式为函数式接口唯一的一个抽象方法add()定义方法体  
MyLambda myLambda=(x,y)->y+x;  
System.*out*.println("两数相加为" + myLambda.add(4,2));  
System.*out*.println("两数相减为" + myLambda.reduce(4,2));

### 7.3.1 Consumer

Consumer是一个函数式编程接口，其特点在于没有返回值，相当于把参数消费了，拥有accept方法和andThen方法。Consumer方法可以接受一个参数，还有一个接受2个参数的BiConsumer

andThen可以连续执行，如下

Consumer f1 = n -> System.*out*.println(n + "-F1");  
Consumer f2 = n -> System.*out*.println(n + "-F2");  
Consumer f3 = n -> System.*out*.println(n + "-F3");  
  
//连续执行F的Accept方法  
f.andThen(f1).andThen(f2).andThen(f3).accept("test");

test

test-F1

test-F2

test-F3

### 7.3.2 Function

### 7.3.3 Supplier

## 7.4 stream流

Stream是java8新特性的最大亮点之一，它和传统意义上的输入输出流完全不同，它是一种流式处理方式，大大加强了java Collection集合对象的数据处理能力。

Stream要求程序员将元素集合看作一种流，集合中的每个元素通过管道流动，我们可以在管道的任意节点上对集合中的每一个元素进行处理， 比如过滤， 排序，聚合等。

Stream 操作包括**中间操作**和**最终操作**。中间操作返回Stream，这样我们就可以在”管道”的任意节点串联多个任意的**中间操作**对流元素进行处理，最后由**终端操作**返回void或者一个非Stream结果值。

因此，**元素流在管道中经过中间操作(intermediate operation)的处理，最后由最终操作(terminal operation)得到处理的结果。**

### 7.4.1 生成streams流

在java8中，jdk为我们提供了两种获取流的方法：

* **stream() − 为集合创建串行流。**
* **parallelStream() − 为集合创建并行流。**

#### 7.4.1.1 stream()生成串行流

串行流的流元素将依次按顺序通过管道，管道中每次只能通过一个流元素，并且按照元素在集合中的顺序依次通过。相比于并行流，串行流不会产生并发问题，因此效率相对于并行流较低，串行流不会改变原本集合中元素的顺序。因此在数据量不大，操作较为简单，对集合元素顺序要求较高的场景，可以使用串行流

除了.stream()方法以外，还有其他方法可以生成串行流，比如：

1.Arrays.stream() ：可以将数组转化为串行流，eg：

**Arrays.stream(new String[10]).collect(Collectors.toList());**

2.Stream.of() ：可以将指定的多个同类型的可变参数转化为串行流，eg:

**Stream.of("aa","bb","cc").collect(Collectors.toList());**

#### 7.4.1.2 parallelStream ()生成并行流

并行流的流元素将并发地通过管道，管道中一次可通过多个元素，实际上可以理解为多线程的stream。相比于串行流，并行流线程不安全，但其效率相对较高，并且不保证原本集合中的元素顺序。在数据量较大，操作较为复杂，对集合元素顺序要求不高的场景，可以使用并行流。当然，java8也提供了相应的方法来对并行流按原集合顺序进行排序的方法。

List<Integer> list=new ArrayList<>();

for (int i=0;i<1000;i++){  
 list.add(i);  
}

//并行流并发执行，limit限制了流元素最多60个，有意思的是，虽然是并发乱序，但最终输出的结果仍 <60   
list.parallelStream().limit(60).forEach(System.*out*::println);

### 7.4.2 中间操作(intermediate operation)

**中间操作**仅仅返回一条全新的被**中间操作**描述过的stream流，实际上并没有执行任何操作，这个特性被称为**”惰性求值”**。也就是说中间操作并不会立即执行，只有最终操作才能触发中间操作。即如果不存在**最终操作**，那么中间操作将不会被执行

#### 7.4.2.1 filter()过滤

filter()操作将对流元素进行测试，通过测试的元素(即返回true)可以通过filter方法，不通过的元素(返回false)将被丢弃，eg：

List<Integer> list=new ArrayList<>();

for (int i=0;i<1000;i++){  
 list.add(i);  
}  
list.stream().filter(integer -> integer<=100).forEach(System.*out*::println);  
list.parallelStream().filter(integer -> integer<=100).forEach(System.*out*::println);

上例只会输出<=100的元素

#### 7.4.2.2 distinct ()去除重复

distinct ()操作将会根据元素的hashcode和equals去除集合中重复的元素，eg：

List<Integer> list=new ArrayList<>();

List<Integer> list\_1=new ArrayList<>();;  
for (int i=0;i<100;i++){  
 list.add(i);  
}  
for (int i=0;i<100;i++){  
 list\_1.add(i);  
}  
list.addAll(list\_1);  
list.stream().distinct().forEach(System.*out*::println);

#### 7.4.2.3 limit ()截断流/skip()跳过元素

limit()操作将返回 Stream 的前面 n 个元素组成新流

skip()操作将跳过(抛弃) Stream 的前n 个元素，返回剩下的元素组成的新流

eg：

List<Integer> list=new ArrayList<>();

for (int i=0;i<50;i++){  
 list.add(i);  
}  
//将返回10-49  
list.stream().skip(10).forEach(System.*out*::println);

//将返回0-9  
list.stream().limit(10).forEach(System.*out*::println);

#### 7.4.2.4 sorted()排序

sorted()操作帮助我们对流元素进行排序（默认是升序），我们也可以利用Comparator对象帮助我们进行扩展排序功能，当然也可以使用lambda表达式简化代码。

Eg：

List<Integer> list=new ArrayList<>();

Random random=new Random();  
for (int i=0;i<10;i++){  
 list.add(random.nextInt(100));  
}

//升序  
list.stream().sorted(new Comparator<Integer>() {  
 @Override  
 public int compare(Integer o1, Integer o2) {  
 return o1-o2;  
 }  
}).forEach(System.*out*::println);

//降序  
list.stream().sorted((x,y)->y-x).forEach(System.*out*::println);

#### 7.4.2.5 map()映射

接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。该操作是中间操作中最为常用的一个操作，eg：

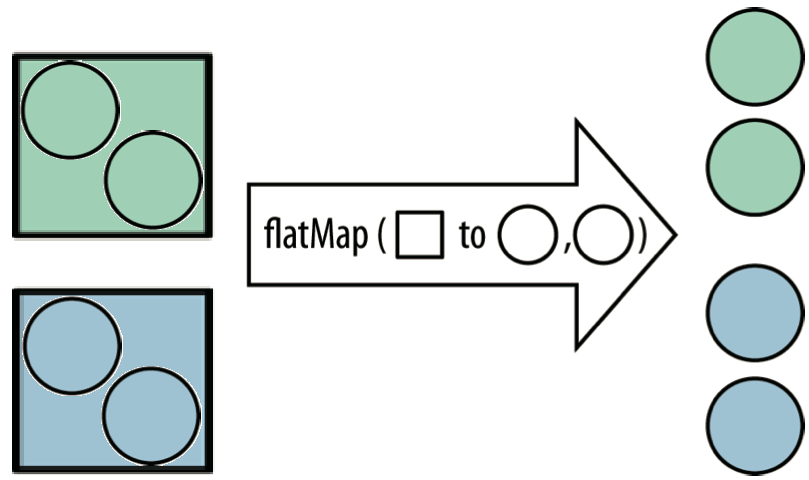
List<Long> pointIds = pointList.stream().map(point ->

Long.*parseLong*(point.getPointId())

).collect(Collectors.*toList*());

#### 7.4.2.6 flatMap()映射

map()操作可以操作单个元素对象，并返回一个被操作后的元素对象，属于一对一映射，这种映射在多数情况下可以满足我们的要求，但当我们想把一个元素对象变为多个元素对象时，map()的表现往往不尽如人意。这时，就轮到flatMap()上场了。如果说map()是一维层次上的一对一映射，那么flatMap()就是将一个二维集合映射为一个一维集合



下面是flatMap()的定义：

将流中的每个元素都映射成一个流，然后把所有流连接成一个流

Eg：

List<String> words=Arrays.*asList*("Hello","World");

List<String> result=words.stream()  
 .flatMap(word -> Arrays.*stream*(word.split("")))  
 .distinct()  
 .collect(Collectors.*toList*());

class MyClass{

private String name;  
 List<Student> studentList;  
 public MyClass(String name){  
 this.name=name;  
 }  
}  
class Student{  
 private String name;  
 public Student(String name){  
 this.name=name;  
 }  
}

class MyClass {

private String name;  
 List<Student> studentList;  
  
 public MyClass(String name, List<Student> studentList) {  
 this.name = name;  
 this.studentList = studentList;  
 }  
  
 public List<Student> getStudentList() {  
 return studentList;  
 }  
}  
  
class Student {  
 private String name;  
  
 public Student(String name) {  
 this.name = name;  
 }  
}

List<MyClass> classesList = new ArrayList<>();  
  
IntStream.*range*(1, 4).forEach(i -> {  
 List<Student> studentList = new ArrayList<>();  
 IntStream.*range*(1, 6).forEach(j -> {  
 studentList.add(new Student("14计科" + i + "班第" + j + "位学生"));  
 });  
 classesList.add(new MyClass("14计科" + i + "班", studentList));  
});  
List<Student> allStudent = classesList.stream().flatMap(  
 myClass -> myClass.getStudentList().stream()  
).collect(Collectors.*toList*());

#### 7.4.2.7 peek()映射

peek()接受一个没有返回值的lambda表达式，实际上，它和map最大的区别就是peek()方法没有返回值，因此peek()方法不会改变流中元素的数据类型，而map()可以，下面是peek()和map()方法的使用，下例中二者的作用是等效的

List<Student> allStudent\_1 = allStudent.stream().peek(

student -> student.setName("我是学生")  
).collect(Collectors.*toList*());  
List<Student> allStudent\_2 = allStudent.stream().map(  
 student -> {  
 student.setName("我是学生");  
 return student;  
 }  
).collect(Collectors.*toList*());

### 7.4.3 最终操作(terminal operation)

一个流只能有一个最终操作，一旦最终操作被执行，那么这个流的所有流元素都被“使用光了”。如果没有最终操作，中间操作也不会执行，最终操作将会触发中间操作

#### 7.4.3.1 foreach()遍历

forEach 方法接收一个 Lambda 表达式，然后在 Stream 的每一个元素上执行该表达式，eg：

**result.stream().distinct().forEach(System.out::println);**

foreach是最终操作，只执行一次就会将流元素消耗殆尽，如果希望遍历输出两次，可以使用中间操作peek()，eg：

Stream.*of*("one", "two", "three", "four")

.filter(e -> e.length() > 3)  
 .peek(e -> System.*out*.println("小写字母: " + e))  
 .map(String::toUpperCase)  
 .peek(e -> System.*out*.println("大写字母: " + e))  
 .collect(Collectors.*toList*());

输出：

小写字母: three

大写字母: THREE

小写字母: four

大写字母: FOUR

#### 7.4.3.2 toArray()转数组

有时候，我们可能希望将stream中的元素转化为一个数组，这时候，就需要用到toArray()方法了，但仅仅使用toArray()方法还不够，我们需要使用方法引用来转化类型

**String[] strArr= list.stream().toArray(String[]::new);**

#### 7.4.3.3 reduce()缩减元素

Reduce中文含义为：减少、缩小，而Stream中的Reduce方法干的正是这样的活，它将对stream中的元素进行计算缩小为一个结果，这个方法存在三个变种：

##### 7.4.3.3.1 一个参数的reduce()

一个参数的reduce将返回一个Optional<T>对象

Optional<T> reduce(BinaryOperator<T> accumulator);

在实际使用时，我们可以使用lambda表达式作为唯一的参数，lambda表达式在这里将接受两个参数，第一个参数是上一次lambda表达式的计算结果(又称中间结果)，第二个参数是下一个stream元素，eg：

Optional<Integer> optional\_maxInt = Stream.*of*(3, 4, 1, 6, 4, 7, 11).reduce((x, y) -> x > y ? x : y);

Optional<Integer> optional\_sum = Stream.*of*(3, 4, 1, 6, 4, 7, 11).reduce((x, y) -> x + y);

上例中reduce方法接受唯一参数，即lambda表达式作为参数

第一个例子中x是上一次三目运算符的计算结果，y则是下一个stream元素

第二个例子中x是上一次x+y的计算结果，y则是下一个stream元素

##### 7.4.3.3.2 两个参数的reduce()

两个参数的reduce的返回值类型和第一个参数保持一致，同样的，我们依然可以使用lambda表达式作为第二个参数

T reduce(T identity, BinaryOperator<T> accumulator);

相比于一个参数的reduce()，两个参数的reduce()返回值不再是Optional类型，并且多了一个初始化的值，也就是说，原本的计算是从stream的第一个元素和第二个元素计算开始的，现在是从给定的第一个参数和stream的第一个元素开始依次向后计算的

Eg：

int b = Stream.*of*(2, 8, 4, 6, 3, 7).reduce(10,(x, y) -> x + y); //10+2+8+4+6+3+7=40

##### 7.4.3.3.3 三个参数的reduce()

三个参数的reduce较为复杂，不常用

#### 7.4.3.4 collect ()

collect()是我们最常用的一种最终操作，它可以将stream流转化为各种不同的集合类型，如List、Set、Map等，我们可以通过内置的Collectors类将stream流转化为以上集合类型，Collectors还有一种分组操作groupingBy，他可以帮助我们按流元素的某个属性值进行分组，他最多可以接受三种参数：

**第一个参数：分组按照什么分类**

**第二个参数：分组最后用什么容器保存返回，默认是HashMap::new**

**第三个参数：按照第一个参数分类后，对应的分类的结果如何收集，默认是Collectors.toList**

1. **一个参数的groupingBy**

将流元素按照该参数进行分组，返回一个map，key是该参数值，value是分组后的各数组

Eg:

List<Student> list= Arrays.*asList*(

new Student(1,"one","zhao"),

new Student(2,"one","qian"),

new Student(3,"two","sun"));  
Map<String,List<Student>> result=list.stream().collect(Collectors.*groupingBy*(Student::getName1));

结果为：

{

one=[

{id=1, name1='one', name2='zhao'},

{id=2, name1='one', name2='qian'}

],

two=[

{id=3, name1='two', name2='sun'}

]

}

1. **两个参数的groupingBy**

将流元素按照第一个参数进行分组，返回一个map

Eg:

List<Student> list= Arrays.*asList*(

new Student(1,"one","zhao"),

new Student(2,"one","qian"),

new Student(3,"two","sun"));

Map<String,Long> result1=list.stream()

.collect(Collectors.*groupingBy*(Student::getName1,Collectors.*toList*()));

Map<String,Long> result2=list.stream()

.collect(Collectors.*groupingBy*(Student::getName1,Collectors.*counting*()));

结果为：

**Result1:**

{

one=[

{id=1, name1='one', name2='zhao'},

{id=2, name1='one', name2='qian'}

],

two=[

{id=3, name1='two', name2='sun'}

]

}

**Result2:**

{one=2, two=1}

1. **三个参数的groupingBy**

将流元素按照第一个参数进行分组，返回一个map，key是第一个参数值，value是第二个参数值

Eg:

List<Student> list= Arrays.*asList*(

new Student(1,"one","zhao"),

new Student(2,"one","qian"),

new Student(3,"two","sun"));

Map<String, Map<String, String>> result = list.stream()  
 .collect(Collectors.*groupingBy*(  
 Student::getName1,  
 HashMap::new,  
 Collectors.*toMap*(Student::getName1, Student::getName2, (x, y) -> x))  
 );

结果为：

{

one={one=zhao},

two={two=sun}

}

#### 7.4.3.5 min()/max()

min()和max()方法接受一个Comparator对象，对流中元素进行比较，最终返回一个唯一的结果，返回结果类型为Optional类型，eg：

int b = Stream.*of*(2, 8, 4, 6, 3, 7).min((x, y) -> x-y).get();//获取最小值

int c = Stream.*of*(2, 8, 4, 6, 3, 7).max((x, y) -> x-y).get();//获取最大值

#### 7.4.3.6 findFirst()/findAny()

在串行流中，findFirst()和findAny()返回的都是串行流中的第一个流元素

在并行流中，findFirst()返回的是串行流中的第一个流元素，而findAny()返回的是流中处理的最快的那个元素。

#### 7.4.3.7 count()

count()方法将返回流中的元素个数，该操作的返回值为long类型

#### 7.4.3.8 anyMatch()/allMatch()/noneMatch

anyMatch表示，判断的条件里，任意一个元素成功，返回true

allMatch表示，判断条件里的元素，所有的都是，返回true

noneMatch跟allMatch相反，判断条件里的元素，所有的都不是，返回true

boolean sdf=list.stream().anyMatch(integer -> integer>0); //true

boolean sdf1=list.stream().allMatch(integer -> integer>0); //false  
boolean none=list.stream().allMatch(integer -> integer>0); //false

## 7.5 Optional<T>类

Optional<T>是一个可选值的包装类，它被用于解决编程中最臭名昭著的问题——空指针异常

先从一个简单的示例开始，在 Java 8 之前，任何访问对象方法或属性的调用都可能导致 NullPointerException

String isocode = user.getAddress().getCountry().getIsocode().toUpperCase();

为了保证上述代码不出现空指针异常，我们常常需要在访问每个值之前对其进行检查：

if (user != null) {

Address address = user.getAddress();  
 if (address != null) {  
 Country country = address.getCountry();  
 if (country != null) {  
 String isocode = country.getIsocode();  
 if (isocode != null) {  
 isocode = isocode.toUpperCase();  
 }  
 }  
 }  
}

如此之多的if嵌套在一起，可读性极差，不利于维护，Optional类的出现则让我们省掉了繁琐的非空的判断

### 7.5.1 常用方法

#### 7.5.1.1 empty()方法

返回一个value为null的Optional对象

public static<T> Optional<T> empty() {

@SuppressWarnings("unchecked")  
 Optional<T> t = (Optional<T>) *EMPTY*;  
 return t;  
}

#### 7.5.1.2 of()方法

of()方法返回一个指定了非null值的Optional对象，如果试图传入一个null，则会抛出异常

Eg：

Optional<String> optional=Optional.*of*("3");

#### 7.5.1.3 ofNullable ()方法

Value值允许为null的of()方法

public static <T> Optional<T> ofNullable(T value) {

return value == null ? *empty*() : *of*(value);  
}

#### 7.5.1.4 isPresent ()方法

如果值存在(value != null)，则返回true，否则返回false

public boolean isPresent() {

return value != null;  
}

#### 7.5.1.5 ifPresent ()方法

ifPresent()方法将接受一个一个Consumer(消费者) 参数，如果对象不是空的，就对执行传入的 Lambda 表达式，lambda表达式的函数返回值为void

public void ifPresent(Consumer<? super T> consumer) {

if (value != null)  
 consumer.accept(value);  
}

eg：

AtomicInteger integer = new AtomicInteger();

optional.ifPresent(i -> {  
 integer.set(Integer.*parseInt*(i));  
});

#### 7.5.1.6 get ()方法

如果Optional中包含值，则返回值。如果Optional为null，抛出异常

public T get() {

if (value == null) {  
 throw new NoSuchElementException("No value present");  
 }  
 return value;  
}

#### 7.5.1.7 orElse()/orElseGet()方法

orElse()给定一个默认值，如果optional对象值为null，则返回默认值，否则返回values

public T orElse(T other) {

return value != null ? value : other;  
}

orElseGet()给定一个无参的lambda表达式作为参数，如果optional对象值为null，则返回默认值(由无参的lambda表达式计算得出)，否则返回values

public T orElseGet(Supplier<? extends T> other) {

return value != null ? value : other.get();  
}

二者在optional为null时，都会返回默认值，但在函数以lambda表达式为参数，并且optional不为null时：

orElse()方法不论optional是否为null，都会执行参数中的函数

orElseGet()方法在optional不为null时，发生类似逻辑短路的机制，不会执行lambda表达式的函数

eg：

public static void main(String[] args) {

List<Integer> list = Arrays.*asList*(10,20,30);  
 //通过reduce方法得到一个Optional类  
 int a = list.stream().reduce(Integer::*sum*).orElse(get("a"));//optional不空时get()方法不会执行  
 int b = list.stream().reduce(Integer::*sum*).orElseGet(() -> get("b"));//optional不空时lambda表达式依然会执行  
 System.*out*.println("a "+a);  
 System.*out*.println("b "+b);  
}  
public static int get(String name){  
 System.*out*.println(name+"执行了方法");  
 return 1;  
}

打印：

**a执行了方法**

**a的值：60**

**b的值：60**

#### 7.5.1.8 orElseThrow()方法

如果值不为null，则取出该值，否则，抛出一个指定的异常(由参数接收)

public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X {

if (value != null) {  
 return value;  
 } else {  
 throw exceptionSupplier.get();  
 }  
}

**eg:**

try {

optional\_1.orElseThrow(NullPointerException::new);  
} catch (Exception e) {  
 e.printStackTrace();  
}

#### 7.5.1.9 map()方法

如果optional不为null，则对其执行mapper函数(value作为mapper的参数)，并将mapper函数的返回值包装成optional，作为map的返回值

public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {

Objects.*requireNonNull*(mapper);  
 if (!isPresent())  
 return *empty*();  
 else {  
 return Optional.*ofNullable*(mapper.apply(value));  
 }  
}

**Eg：**

class Class {

private Teacher teacher;  
 public Teacher getTeacher() {return teacher;}  
 public void setTeacher(Teacher teacher) {this.teacher = teacher;}  
}  
  
class Teacher {  
 private String name;  
 public String getName() {return name;}  
 public void setName(String name) {this.name = name;}  
}

Class myclass = new Class();

Teacher teacher = new Teacher();  
teacher.setName("张三");  
myclass.setTeacher(teacher);

Optional<Class> optionalInteger = Optional.*ofNullable*(myclass);  
String teacherName = optionalInteger.map(Class::getTeacher).map(Teacher::getName).get();//张三

#### 7.5.1.10 flatMap()方法

相比于map()，flatMap()与其相比最大的区别在于二者的参数，二者都接受一个lambda表达式作为参数，map()接受的lambda表达式返回值可以是任意值，而flatMap()的lambda表达式必须返回Optional类型，eg：

String teacherName=optionalInteger

.flatMap(myClass -> Optional.*of*(myClass.getTeacher()))

.flatMap(myTeacher -> Optional.*of*(myTeacher.getName()))  
 .get();

# 8.动态代理技术

# 9.位运算

常见的位运算有6种：

1. & 与运算
2. | 或运算
3. ^ 异或运算
4. ~ 取反运算
5. << 左移运算
6. >> 右移运算

## 9.1 & 与运算

& 与运算 两个位都是 1 时，结果才为 1，否则为 0，

eg：

**1 0 0 1 1 &**

**1 1 0 0 1**

**------------------------------**

**1 0 0 0 1**

**数学涵义：对n取模 等价于 对n-1按位与（前提是n是2^n）**

**即： x % n == x & (n-1) 或**

**x & n == x % (n+1)**

**为什么会有这样的数学涵义呢？因为2的n次方实际就是1后面n个0，2的n次方-1 实际就是n个1，对n个1进行&操作，就是求其余数，这也是hashmap的桶大小为什么一定要是2^n的原因，因为这样的话，能最大程度较少hash碰撞发生的概率，尽量做到均匀分布**

## 9.2 | 或运算

两个位都是 0 时，结果才为 0，否则为 1

eg：

**1 0 0 1 1 |**

**1 1 0 0 1**

**------------------------------**

**1 1 0 1 1**

## 9.3 ^ 异或运算

^ 异或运算，两个位相同则为 0，不同则为 1

eg：

**1 0 0 1 1 ^**

**1 1 0 0 1**

**-----------------------------**

**0 1 0 1 0**

## 9.4 ~ 取反运算

~ 取反运算，0 则变为 1，1 则变为 0

eg：

**~ 1 0 0 1 1**

**-----------------------------**

**0 1 1 0 0 ​**

## 9.5 << 左移运算

<< 左移运算，向左进行移位操作，高位丢弃，低位补 0

eg：

**int a = 8;**

**a << 3;**

**移位前：0000 0000 0000 0000 0000 0000 0000 1000**

**移位后：0000 0000 0000 0000 0000 0000 0100 0000**

**数学涵义：左移n位 等价于 ×2^n**

**即： x << n == x × 2^n**

## 9.6 >> 右移运算

>> 右移运算，向右进行移位操作，对无符号数，高位补 0，对于有符号数，高位补符号位，eg:

**unsigned int a = 8;**

**a >> 3;**

**移位前：0000 0000 0000 0000 0000 0000 0000 1000**

**移位后：0000 0000 0000 0000 0000 0000 0000 0001**

**​**

**int a = -8;**

**a >> 3;**

**移位前(-8)：1111 1111 1111 1111 1111 1111 1111 1000**

**移位前(-1)：1111 1111 1111 1111 1111 1111 1111 1111**

**数学涵义：右移n位 等价于 /2^n**

**即： x >> n == x / 2^n**

# 5.异常

## 5.1 简介

异常是java代码的错误，但并不是所有错误都是异常，java有如下三种异常：

**1.检查性异常**：最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。所有检查性异常都继承自**java.lang.Exception**

**这类异常被要求必须处理(try-catch、throws)**

eg：输入输出异常(IOException)、文件不存在异常(FileNotFoundException)、SQL语句异常(SQLException)等

**2.非检查性异常(运行时异常)**： 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。所有检查性异常都继承自**java.lang.RuntimeException。**

**这类异常不强制要求处理，但也可以被捕获处理**

eg：空指针异常(NullPointerException)、除零异常(ArithmeticException)、数组越界异常(ArrayIndexOutOfBoundsException)等

**3.错误**： 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

