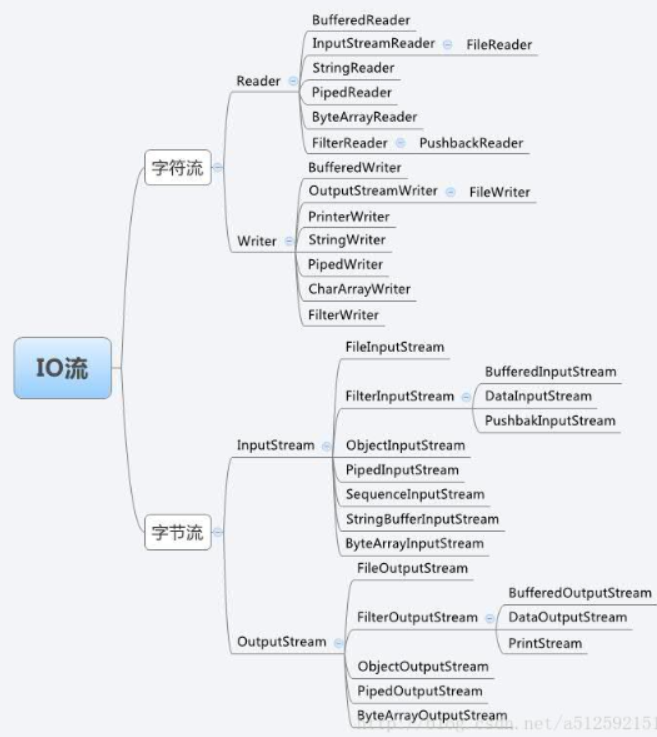
## IO 和NIO相关

I/O问题可以说是当今Web应用中所面临的主要问题之一，因为在当前这个海量数据时代，数据在网络中随处流动。在这个流动过程中都涉及到IO问题，可以说大部分Web应用系统的瓶颈都是IO瓶颈。

### JAVA中的IO基本架构

1. 基于字节操作的io接口: InputStream 和 OutputStream。
2. 基于字符操作的io接口:Writer和Reader。
3. 基于磁盘操作的io接口:File。
4. 基于网络操作的io接口:Socket。

1和2主要是数据传输的格式，后两组主要是数据传输的方式。

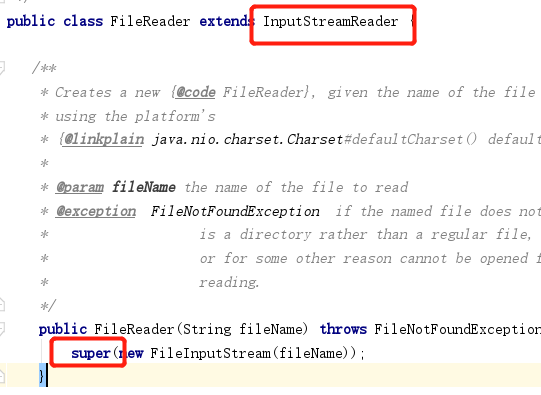
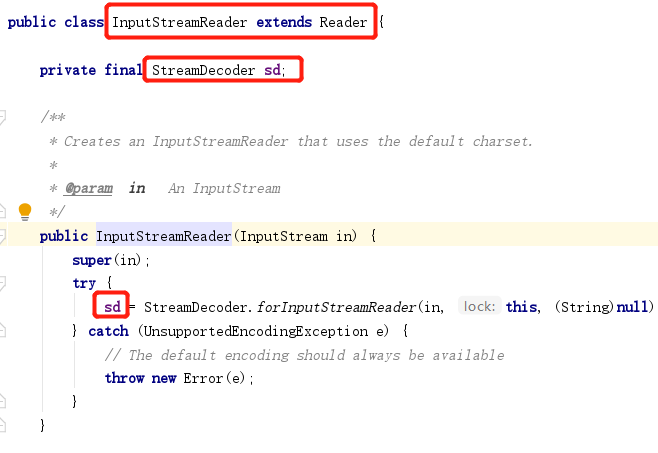


除此之外，还可以分为节点流(低级流)和高级流。

低级流：可以从或向一个特定的地方（节点）读写数据。如FileReader。

高级流：是对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。如BufferedReader.处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。如：InputStreamReader OutputStreamReader BufferedInputStrean BufferedOutputStream BufferedReader BufferedWriter DataInputStream DataOutputStream。

\* 字节与字符的转化接口：数据持久化或网络传输都是以字节进行的，所以必须要有从字符到字节或从字节到字符的转化。InputStreamReader类是从字节到字符的转化桥梁，从Inputtream到Reader的过程要指定编码字符集，否则将采用系统默认的字符集，很可能出现乱码，StreamDecoder正式从你字节到字符的解码的实现类。例如，FileReader继承了InputStreamReader类，实际上是读取文件流，然后通过StreamDecoder解码成char。

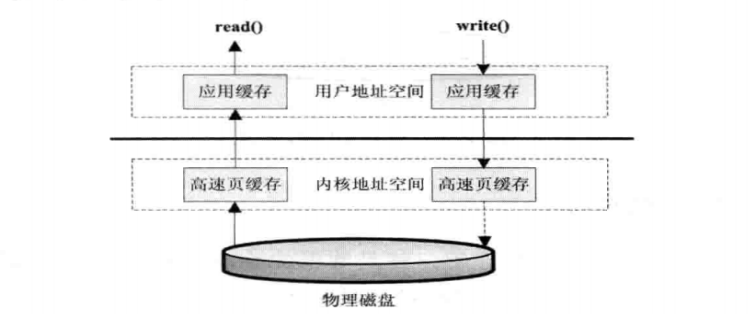
### 磁盘IO工作机制

应用程序要访问物理设备只能通过系统调用的方式来工作，读和写分别是read()和write()两个系统调用。而只要是系统调用就可能存在内核空间地址和用户空间地址切换的问题，这是操作系统为了保护系统本身的运行按权，而将内核程序运行使用的内存空间和用户程序运行的内存空间进行隔离造成的。这样保证了内存程序运行的安全性，但是也必然存在数据可能需要从内存空间想用户空间复制的问题。，为了加速io访问，在内核空间使用缓存机制，也就是将从磁盘读取的文件按照一定的组织方式进行缓存，如果用户访问的是同一段磁盘地址的空间数据，那么操作系统将从内核缓存中直接取出返回给用户程序，这样可以减少io响应时间。

#### 几种访问文件的方式：

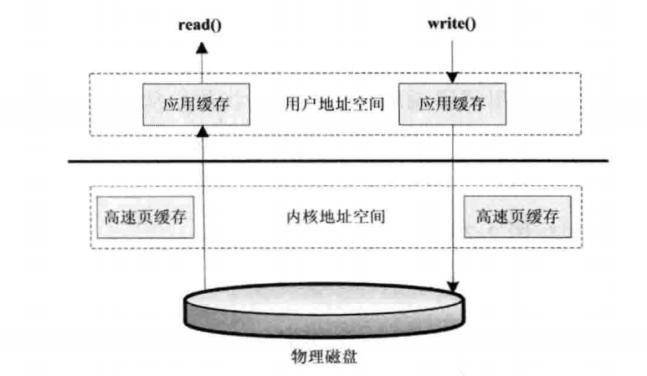
1. 标准访问文件的方式  
    当应用程序调用read()接口时，操作系统检查在内核的高速缓存中有没有需要的数据，如果已经缓存了，那么直接从你缓存中返回，如果没有，则从磁盘读取，然后缓存在操作系统的缓存中。

当应用程序调用write()接口将数据从用户地址空间复制到内核地址空间的缓存中。这时对用户程序来说，写操作就已经完成，至于什么时候写道磁盘由系统决定，除非显式调用sync同步命令。



1. 直接IO的方式，

直接访问磁盘，不经过内核数据缓冲区，减少从内核缓冲区到用户程序缓存的数据复制。



1. 同步访问文件的方式

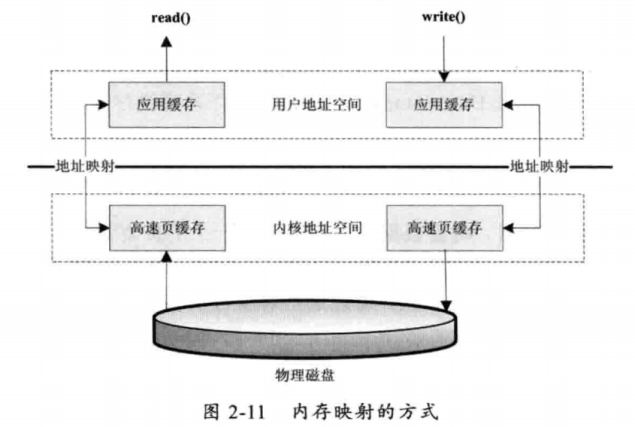
数据的读取和输入都是同步操作的，他与标准访问文件的方式不同，只有当数据被成功写到磁盘时才返回给应用程序成功的标志。此方式只在一些对数据安全性要求比较高的场景中才使用

1. 异步访问文件的方式

访问数据的线程发出请求之后，线程会接着处理其他事情，而不是阻塞等待，当请求的数据返回后继续处理下边的操作。可以明显提高应用程序的效率，不会改变访问文件的效率。

1. 内存映射的方式

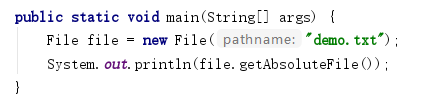
将内存中的某一块区域与磁盘中的文件关联起来，当要访问内存中的一段数据时，转换为访问文件的某一段数据。目的同样是减少数据从内核空间缓存到用户空间缓存的数据复制操作。



#### Java访问磁盘文件

数据在磁盘中的唯一最小描述就是文件，也就是说上层应用程序只能通过文件来操作磁盘上的数据，文件也是操作系统和磁盘驱动器交互的最小单元。在Java中通常的File并不代表一个真实存在的文件对象，创建File时，会返回一个代表这个路径的虚拟对象。

在真正读取这个文件时，才会检查这个文件存不存在。

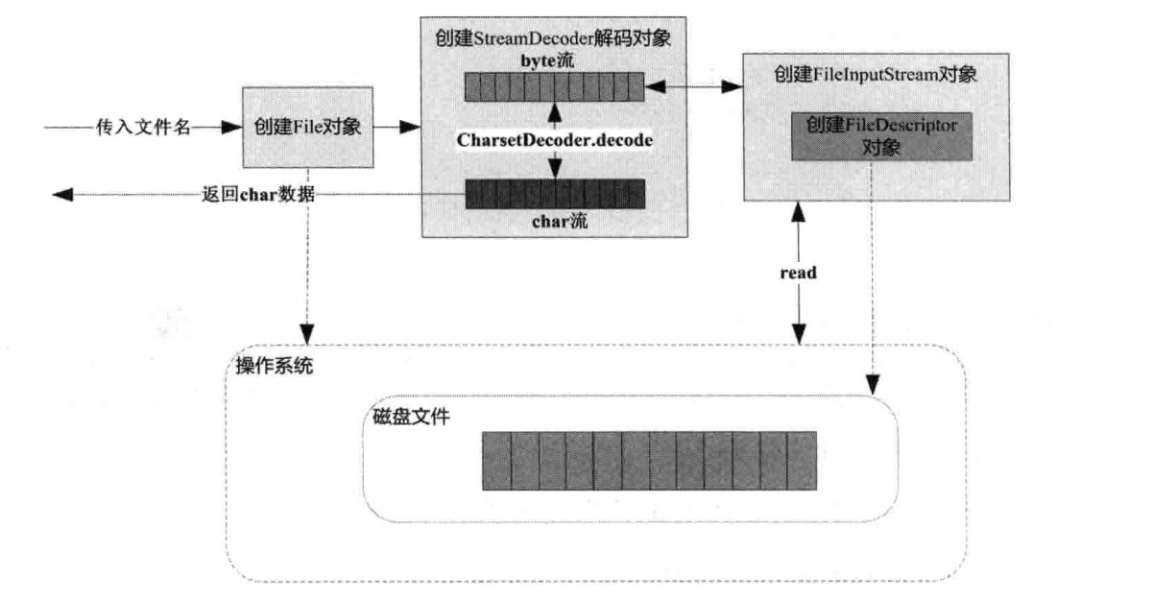


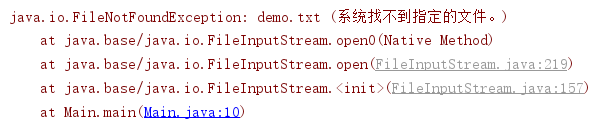
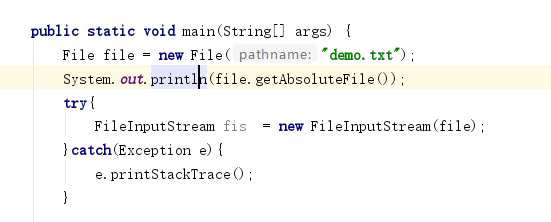
此时，并没有demo.txt文件，但仍能正常使用file.api。

只有在真正读取这个文件时，才会真正检查一个文件存不存在。例如FileInputStream类都是操作一个文件的接口，在创建一个FileInputStream对象时会创建一个FileDescriptor对象，这个对象就是真正代表一个存在的文件对象的描述。

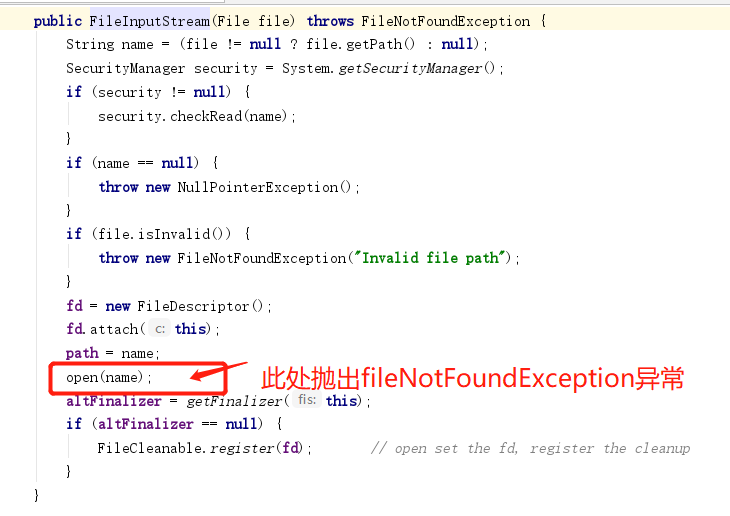
！现在以读取文件的程序为例，介绍如何从磁盘读取一段文本字符。

当传入一个文件路径时，将会根据这个路径创建一个File对象来表示这个文件，再根据File对象创建真正读取文件的操作对象，这是将会创建一个关联真是存在的磁盘文件的文件描述符FileDescriptor，通过这个对象可以直接控制这个磁盘文件，由于我们需要读取的时字符格式，所以需要StreamDecoder类将byte解码为char格式。





此时会报错。



#### Java序列化技术

Java序列化就是将一个对象转化成一串二进制表示的字节数组，通过保存或转移这些字节数据来达到持久化的目的。需要持久化，对象必须继承java.io.Serializable接口，反序列化则是相反的规则，将这个字节数组再重新构造成对象。序列化的数据并不像class文件那样保存类的完整的结构信息。

序列化一些总结：

1. 当父类继承Serializable接口时，所有子类都可以被序列化。
2. 子类实现了Serializable接口，父类没有，父类中的属性不能序列化(不报错数据会丢失)， 但是在子类中属性仍能正确序列化
3. 如果序列化的属性是对象，则这个对象也必须实现Serializable接口，否则会报错。
4. 在反序列化时，如果对象的属性有修改或删减，则修改的部分属性会丢失，但不会报错。
5. 在反序列化时，如果serialVersionUid被修改，则反序列化会失败。

### 网络IO工作机制

#### TCP状态转化

1. TCP建立链接需要3次握手，详情如下：

相关状态解释：

CLOSED：虚拟出来的状态，实际不存在，你在netstat -ant | grep 端口的时候是找不到的；

LISTEN：表示Server大门已开，随时准备有Client前来连线；

SYN\_SENT：只在Client端出现，表示Client发送过SYN了，正在焦急地等待Server的ACK；

SYN\_RCVD：只在Server端出现，表示Server收到Client的SYN了，并且已经发给Client自 己的ACK和SYN了，正在焦急地等待Client的ACK；

ESTABLISHED：在Client端出现表示Client把自己的ACK（第3次握手）发出去了，Client已 经就绪；在Server端出现表示Server已经收到Client的ACK（第3次握手）了，Server 已经就绪；

注：只有在Client和Server同时为ESTABLISHED时，即同时就绪时才可以进行数据传输。

Client端口状态转换：

1. 应用层调用connect，发送SYN到对端，等待对端的ACK和SYN；

2. 等待对端的ACK和SYN到来，接收到ACK和SYN后，发出自己的ACK，状态进入ESTABLISHED；

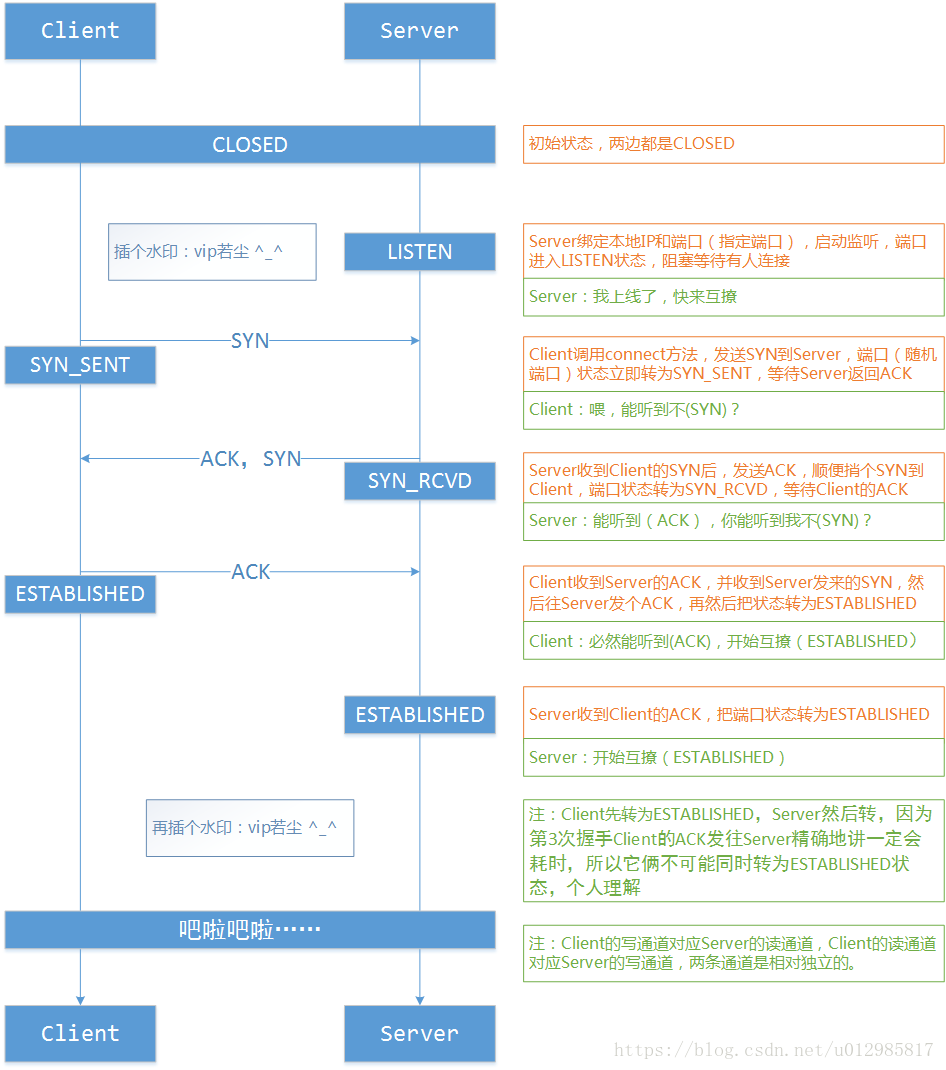
3. 等待对端的ACK和SYN期间，端口状态一直为SYN\_SENT，超时后进入CLOSED；

Server端口状态转化：

1. 应用层启动侦听，端口进入LISTEN状态；

2. 接收到Client发来的SYN，发送自己的ACK和SYN，进入SYN\_RCVD状态，等待Client的ACK；

3. 等待Client的ACK到来，接收到ACK后，进入ESTABLISHED；等待超时，进入CLOSED；



2 Tcp断开链接：

此过程需要进行4次挥手

状态解释：

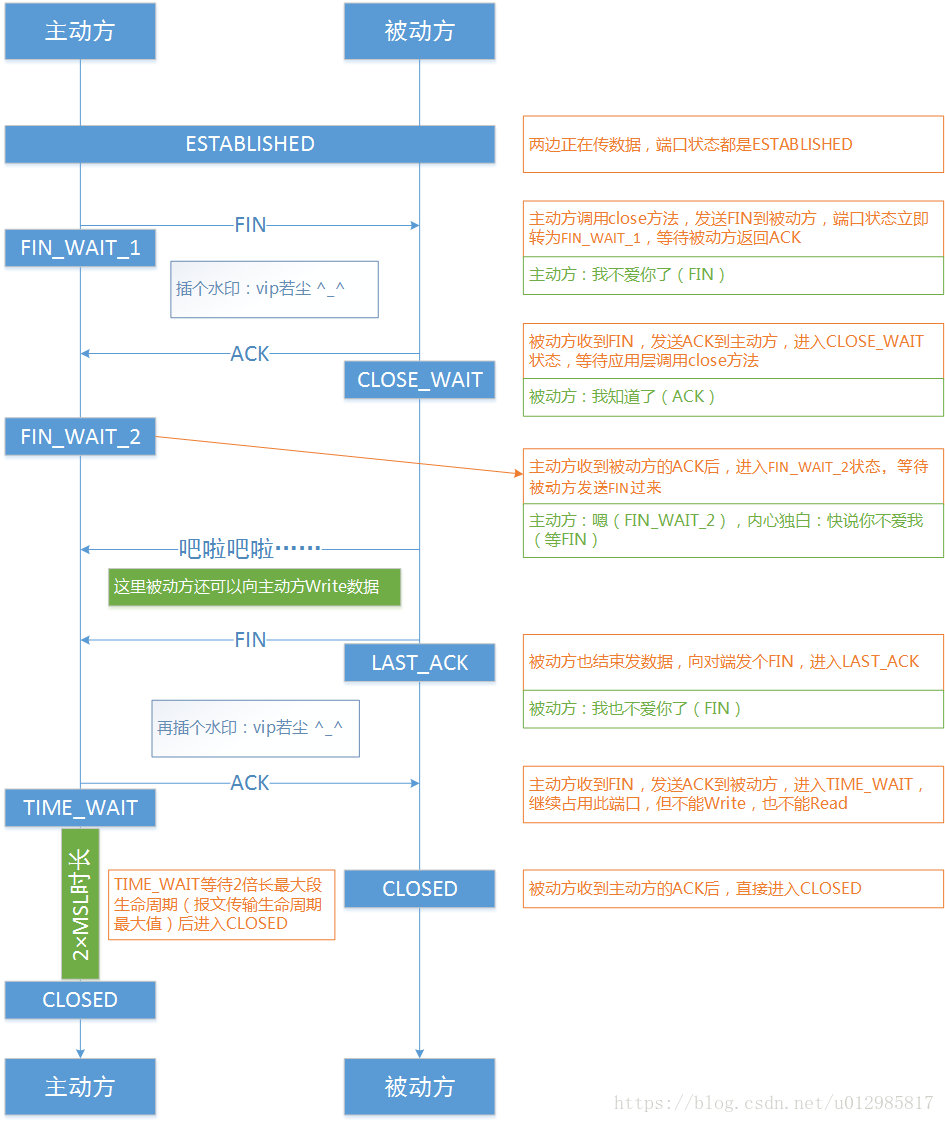
FIN\_WAIT\_1：仅出现在主动方，表示主动方想要断开连接，已经关闭了写通道，并向对端发送了FIN，等待对方的ACK到来

CLOSE\_WAIT：仅出现在被动方，表示被动方收到FIN后，已经回复ACK，正在等自己的应用层调用close方法关闭写通道，在CLOSE\_WAIT状态下，自己只能发送数据，但不能接收数据；

FIN\_WAIT\_2：仅出现在主动方，表示主动方已收到对端的ACK，等待对端的FIN，此时无法再发送数据，但是可以接收数据；

LAST\_ACK：仅出现在被动方，表示被动方缓冲区数据已经发送完毕，并且已经发送FIN到对端，等待对端的ACK；如果应用层写的比较垃圾，没有调用close关闭socket，则会一直停留在CLOSE\_WAIT；

TIME\_WAIT：仅出现在主动方，表示主动方已经发出ACK了，本次通讯完事了，双方都不能再读写了，但是主动方不确定对方能不能收到最后一个ACK，为了保证这个端口释放后，不被后来的连接马上使用被当成是新连接，通俗地讲，为了不乱套，这个状态会一直等待，等多久呢， 等2×MSL（Maximum Segment Lifetime）个时间，这个MSL是操作系统配置的，有默认参数，可以改。



注：TIME\_WAIT:这个状态比较复杂。有3个状态可以转化为此状态

1. 由FIN-WAIT-2转换到TIME-WAIT，具体情况时，在双方不同是发起FIN的情况下，主动关闭的一方完成在自身发起的关闭请求后，接受到被动关闭乙方的fin后进入的此状态。(正常情况)
2. 有一种ClOSING状态：两边同事发起关闭请求，由FIN-WAIT-1进入此状态，具体动作是接收到FIN请求，同事响应一个ACK。正常情况下，当你发送FIN报文后，按理来说是应该先收到（或同时收到）对方的ACK报文，再收到对方的FIN报文。但是CLOSING状态表示你发送FIN报文后，并没有收到对方的ACK报文，反而却也收到了对方的FIN报文。

从CLOSING转换到TIME-WAIT，具体情况是：在双方同时发起关闭，都做了发起FIN的请求，同时接收到了FIN并作了ACK的情况下。由CLOSING状态进入此状态。

1. 由FIN-WAIT-1转换到此状态，具体情况是：同时接受到FIN(对方发起)和ACK(本身发起的FIN回应)，它与CLOSING转换到TIME-WAIT的区别在于本身发起的FIN回应的ACK先于对方的FIN请求到达。而由CLOSING转化则是FIN先到达