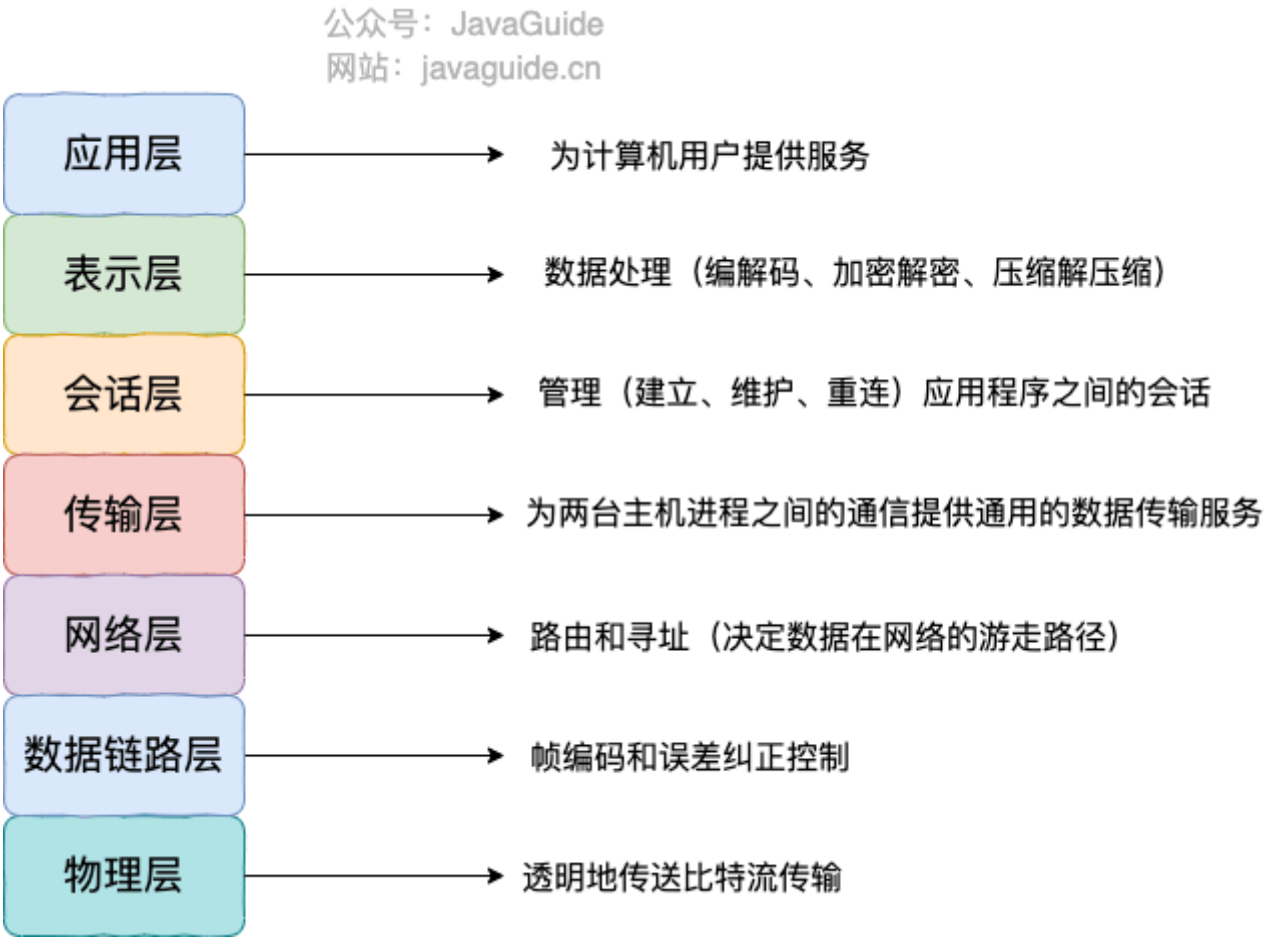
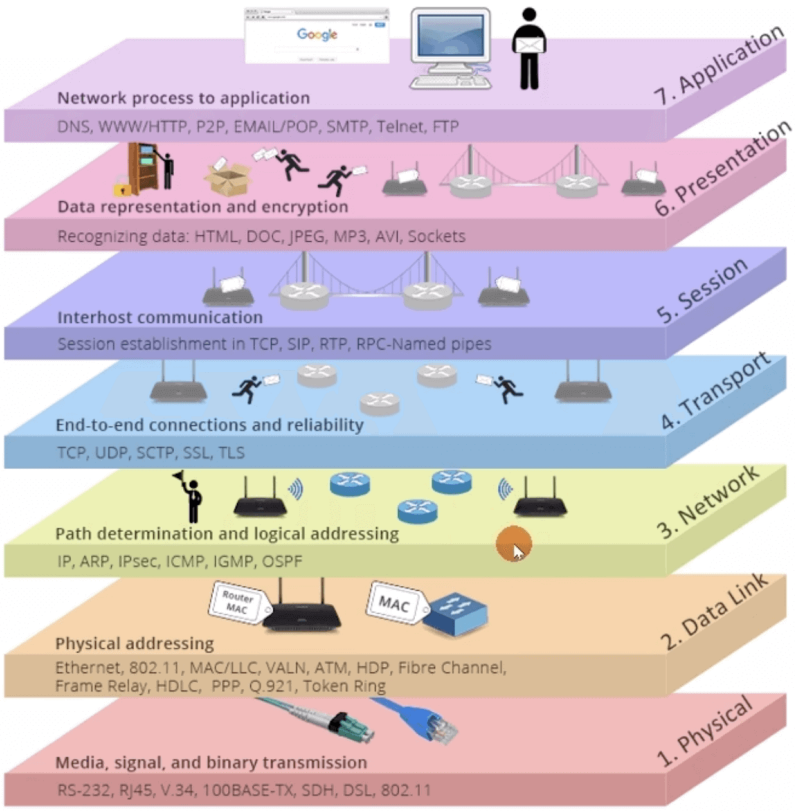
网络

基础知识

OSI7层

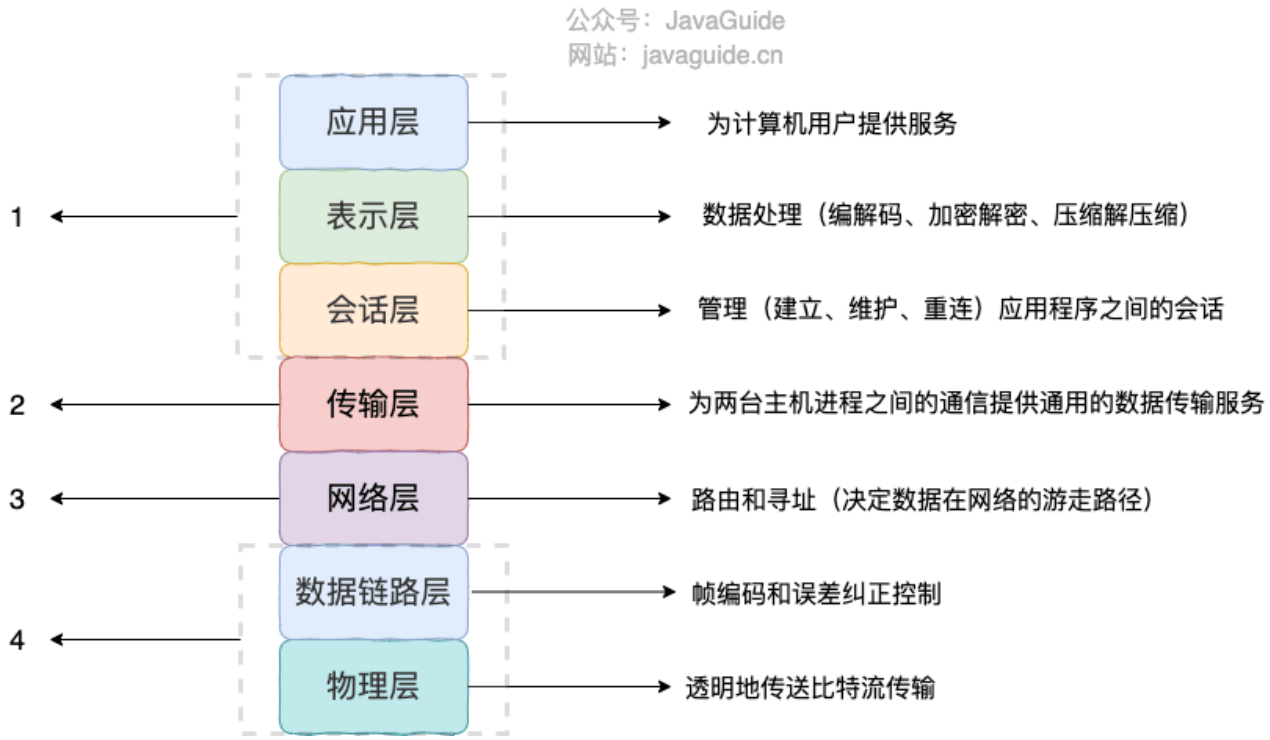




- 应用层
- 表示层
- 会话层
- 传输层
- 网络层
- 数据链路层
- 物理层

TCP/IP 四层模型

1. 应用层
2. 传输层
3. 网络层
4. 网络接口层



> 应用层

应用程序的服务

HTTP SMTP POP3 IMAP FTP SSH DNS域名管理系统

> 传输层

TCP UDP

> 网络层

IP ARP ICMP NAT OSPF BGP RIP

> 网络接口层

MAC 以太网 差错检测

HTTP

> 报文结构

请求行：GET POST PUT DELETE ..... 请求方法+URL+ 协议版本号

请求头：字段

空行

请求体：实际数据

> 响应报文

状态行：状态码 状态信息 版本

响应头部

空行

响应体

> 从输入*URL*到页面展示到底发生了什么？

浏览器输入URL ---**DNS**协议拿IP地址-----**IP**+端口 发**TCP**请求-----在TCP连接上 发送**HTTP**请求报文 获取网页内容------服务器收到请求报文，返回相应报文-----浏览器收到相应报文，解析数据结构，如果还要资源要加载则继续来回发和收报文 直到网页显示-----不需要和服务器通信可主动关闭TCP连接

> 用到什么协议？

加上OSPF和ARP   前者是开放最短路径优先协议 后者是IP地址转MAC地址

DNS:简单讲是逐级往上 要IP映射   找到了就返回本地服务器留作以后用

> *HTTP*状态码？

1xx 正在处理   2xx 成功   3xx重定向 4xx 服务处理请求失败（服务器是正常的）   5xx服务器异常

301永久重定向 302临时重定向   404无法找到此页面

> *HTTP*字段

跳了 太多了 cookie Accept:能够接受的回应内容类型（Content-Types）。

> *Http*和*Https*有什么区别 *http*不安全？

HTTP 80端口 HTTPS是443端口

url前缀不一样（显而易见）

HTTP 协议运行在 TCP 之上，所有传输的内容都是明文，客户端和服务器端都无法验证对方的身份。HTTPS 是运行在 **SSL/TLS** 之上的 HTTP 协议，**SSL/TLS** 运行在 **TCP** 之上。所有传输的内容都经过加密，加密采用对称加密，但对称加密的密钥用服务器方的证书进行了非对称加密。

TCP建立连接三次握手后还有握手过程 密钥交换

传输信息 对称加密 传 数据传输用的对称加密的秘钥 用非对称

如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密。因为加密和解密使用的是两个不同的密钥，所以这种算法叫作非对称加密算法。

> *SSL/TLS*的握手过程？

TCP建立三次后 还有四次秘钥握手

第一次：发起请求 发送 版本信息 客户端随机数 密码套件

第二次：服务器响应 确认版本 服务器的随机数 确认密码套件 服务器的数字证书

第三次：客户端确认证书安全性，并从中取出公钥（服务器的公钥） 用公钥加密

发送 新随机数（被服务器的公钥加密） 加密算法改变通知（后面用对称加密） 握手结束通知

两边都有同样的三个随机数 来合并成会话秘钥

第四次：服务器解密被公钥加密的随机数 并计算生成会话秘钥（） 加密算法改变通知（后面用对称加密） 握手结束通知

TLS 握手完成后，客户端和服务器的会话密钥是相同的。

> 称加密的密钥用服务器方的证书进行了非对称加密。

关键点在于 TLS的公钥是服务器的　所以其实就是 客户端用公钥加密第三随机数　服务器用私钥+公钥解密

> *GET*和*POST*的区别

GET是获取资源 幂等操作 安全所以缓存

POST是根据请求负荷对指定的资源做出处理 提交或修改资源 非幂等的　不安全所以不缓存

> 长连接

每次请求都要建立TCP和释放连接 那就是短连接

长连接 HTTP 的 Keep-Alive 就是实现了这个功能 ： TCP建立连接后不需要反复建立连接和释放 长期在

> *HTTP1.1*的拆包？

请求头会有content length字段　表示正文字节数 然后就会根据这个来取对应值的字节

> *Http1.1*和*1.0*区别？

1.1长连接 1.0 默认短连接

1.1多了很多状态码

HTTP1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，节省资源

Host 头处理 : HTTP/1.1 在请求头中加入了`Host`字段 域名系统（DNS）允许多个主机名绑定到同一个 IP 地址上，但是 HTTP/1.0 并没有考虑这个问题 （多了个标签） 虚拟主机

> *1.1*和*2.0*区别？

2.0加入头压缩 相当于一个hashmap/hashset　相同头的只发索引号

2.0采用二进制格式，统称为帧 增加数据传输效率

多路复用 解决应用层的队头阻塞

服务器可以主动发消息

> *2.0*和*3.0*的区别

3.0新增了 QUIC　UDP协议的升级版 QUIC 建立连接握手次数是两次

解决了**TCP**的队头阻塞问题 使用类似NIO的解决方法 多路复用数据流

整个报文都进行加密

连接迁移，64位ID就不会中断

> 什么时候会中断

四次握手

超时重传最大次数会断开

长时间没有请求和响应 到达阈值会释放

> *HTTP TCP SOCKET*区别

HTTP 应用层协议 用于传输超文本数据 定义数据格式和规则

TCP面向连接的可靠的传输层协议 负责建立可靠的数据传输连接

Socket是 通信抽象接口，

>

DNS

域名转IP

越靠右层级越高

> 解析流程？

请求 先在本地找 然后逐级往上找 找到了再返回到本地服务器

> 端口

53

> 底层？

UDP 快速响应 占用小 拿不到就拼命发就好了

Cookie和Session

> *HTTP*到底是不是无状态？

无状态，每个请求都是独立的 可以用cookie和session来跟踪状态

> 携带*Cookie*的*http*究竟是有状态还是无状态

一定程度上有状态 因为可以识别用户状态啥的

> *cookie*和*session*的区别是？

1.cookie在客户端 session在服务端 每个用户分配一个session id 然后会存到cookie 后面请求会带上session id

2.数据容量 单个Cookie在4kb左右 session理论没限制

3.cookie不安全 session更安全

4.生命周期：Cookie可以设置过期时间 session默认关闭浏览器时session结束 但是服务器也可以设置session超时时间

5.性能：没啥好说的 cookie影响网络效率 session增加服务器负载

> *token*又是什么？

token类似令牌 无状态 用户信息被加密到token中 服务器解密 知道是哪个用户

> 禁用*cookie session*能用吗？

默认情况下不能用 因为session id发到客户端后 要存到cookie 这样后续请求才能带着session id去访问

两个方法：

1.url重写 ：session id加到url作为参数

2.隐藏表单字段:html表单中包含隐藏字段，用来存session id （其实还要存 只不过绕过cookie）

> *localStorage*和*Cookie*有什么区别

localStorage更大

localStorage只存不发

localstorage永久存

相对安全 因为不会发布出去

JWT

> 组成部分

- 头部（Header）：这部分通常是一个JSON对象，描述了JWT的签名算法和其它元数据。 Base64序列化
- 有效载荷（Payload）：这部分也通常是一个JSON对象，包含了发行方信息、用户信息、过期时间等声明。Base64序列化

- 签名（Signature）：这是header和payload的数字签名，使用header中指定的签名算法和秘钥生成，用于验证JWT的完整性和真实性。

> 和传统方式有什么区别

无状态：不会在服务器端存会话信息　jwt包含了所有必要的信息 用户身份 权限

安全性：使用秘钥对令牌进行签名，确保完整和真实性　只有持有正确秘钥的服务器才能解析令牌

跨域支持：可以在不同域之间传递

> *JWT*令牌为什么能解决集群部署　什么是集群部署

传统方式不能用在分布式 因为分布式服务器不共享会话信息 只能引入redis （增大开销）

jwt无状态且可以跨域　用户登录后就会返回一个jwt令牌　客户端就会带着jwt令牌访问

> 缺点

派发出去没办法撤销

只能加黑名单机制(redis)

> 泄露了怎么解决？

及时失效：检测到风险就立即失效

刷新令牌：检测到泄露 可以主动刷新令牌，旧令牌标记为失效

访问令牌失效后， 客户端会使用刷新令牌向服务器请求一个新的访问令牌。服务器会验证刷新令牌是否有效，检查它是否过期或是否已经被撤销。如果刷新令牌有效，服务器会生成新的访问令牌并返回给客户端。

说到底还是黑名单

> *jwt*存哪？

存localstorage: 缺点有XSS风险　优点 存储空间大 也不会出现在请求中（快）

存Session storage：窗口关闭（会话关闭）就没了 要重新请求jwt

存Cookie: 可以设置httponly防止JS访问 和利用secure标志确保仅通过http发送　缺点是cookie小 而且受CSRF攻击风险 请求都会带上 太慢

存客户端

其他

> *HTTP*长连接和*WebSocket*区别

TCP全双工 http半双工　websocket全双工

频繁交互场景用websocket　http1.1 不问就不答（半双工 数据在两个方向之间可以流动，但不能同时进行双向通信。）

Nginx

在应用层

> 负载均衡算法

轮询 IP哈希 （指定服务器） URL哈希 （指定服务器）　最短响应时间（先Ping后发） 加权轮询

传输层、UDP、TCP

> 头部

20字节 无选项

源端口 目标端口

序列号　解决顺序问题

确认应答号：解决丢包问题

控制位：ACK　确认应答　RST异常断开　SYN希望建立连接　FIN 希望断开连接

窗口大小 接收端还能接收多少数据

数据偏移 指定 **TCP** 头部的长度

> *udp*头部

八字节

16位源端口 16位目标端口 16位长度 16位检验和

> 三次握手

第一次 客户端发SYN=1 序列号假设为k

第二次 服务端发 ACK=1 SYN=1 确认Ack号为K+1 序列号为n 第二次是合并了syn和ack

第三次客户端发 ACK=1 ACk=n+1 可携带数据

> *Why* 三次握手

阻止重复连接 同步初始序列号 避免资源浪费（如果只有两次握手 多发了syn报文 导致新建了多个连接）

> 为什么阻止重复连接

因为客户端以最新发的syn报文 序列为base 收到 ack为该序号+1 才确认有效

> 第三次握手 确认保丢失会发生什么？

如果丢失 服务端在等第三次握手 迟迟收不到确认 就会重发syn ack （重新第二次握手）

ack报文本身不会重传

> 客户端和服务端状态是？

第一次握手后 客户端为syn-sent状态

第二次握手发出（接收第一次握手报文后） 服务端为syn_rcvd状态

第三次握手 客户端发出报文后进入established状态 服务端收到后也变成established状态

> *accept*是什么？

accept是java socket库的方法

调用accept 从把连接取出来给用户程序使用

第一次握手 会把连接存到半连接队列　收到第三次后会移除对应半连接 然后创建新链接放到全连接队列

存的TCB控制块 有端口IP 序列号等信息 超时时间 （问下去就说不会啦）

> 大量*SYN*包（第一次握手）冲到服务器会发生什么？

半连接队列会爆　收到就丢弃了

调大max backlog　网卡接收速度比内核处理快 就放到这个队列

增加半连接队列大小

开启syncookies功能 而是根据 `SYN` 生成一个特殊的序列号（**Cookie**），并返回给客户端，然后直接丢弃连接状态。

- 如果客户端是真实的，它会回复 `ACK`，其中包含这个 Cookie。
- 服务器检查 `ACK` 里的 Cookie，如果有效，则认为连接有效，直接建立完整连接，不需要 `SYN Queue` 记录。

会丢弃数据 少用

减少 SYN-ACK 的重传次数，以加快处于 SYN_REVC 状态的 TCP 连接断开。

> *TCP* 四次挥手*(*关闭连接*)*

第一次 客户端发FIN报文 通知要关闭 自己进入fin_wait_1状态　服务端收到后转为CLOSE_WAIT状态 缓冲区加入EOF （终止符）

第二次 服务端发送ACK　客户端收到变成 FIN_wait状态

服务端读完缓冲区数据 并读到EOF 调用关闭函数 则第三次挥手 FIN报文 进入Last_ack状态

客户端收到后 进入Time_wait状态

客户端返回ACK（第四次） 服务器收到后关闭

客观客户端最后要等2MSL再关闭

> 为什么第二次和第三次没有合并？

因为服务端缓冲区还有数据没接收完 确保数据完整 等到读到EOF 缓冲区接收完毕才能通知客户端要关闭了

主动方在服务器一端

如果服务端（被动方） 没有数据并且开启了TCP延迟确认机制 那么第二和第三就能合并

> 如果第三次握手一直没发？

如果是shutdown 那没啥事 卡在fin_wait2问题不大

如果是close函数，客户端会一直卡在fin_wait2状态 这个状态没意义 可以设置 **tcp_fin_timeout** 超时时间 默认值是60秒 超过60秒会直接关闭

> *shutdown*和*close*的区别？

shutdown可以选择关闭方向 可以继续接收数据或发数据

close则直接关闭 啥都没了

> 如果第一次握手失败或客户端发*FIN*包丢失？

第一次握手丢了 迟迟收不到ACK 会触发超时重传 次数超过阈值 会等两倍时间 还是收不到 就直接close状态了

> 为什么要等到*2MSL*

确保最后的 ACK 能够被服务器接收

如果客户端的最后一个 `ACK` 丢失了，服务器会重传 `FIN`。

防止旧连接数据影响新连接

> 服务端出现大量*time-wait*原因？

HTTP没有长连接（提示 1.1默认长连接）

HTTP长连接超时，HTTP长连接可在同一个TCP上发和接多个HTTP（类似NIO），但是不能占着，有keepalive_timeout参数 超过时间就关闭TCP连接

HTTP长连接请求数量达到上限　keepalive_requests参数默认值是100 小于实际场景

> *TCP*和*UDP*的区别？

TCP是面向连接的 UDP不需要连接

TCP是一对一 UDP是一对多 或多对一

TCP是可靠的 UDP不可靠 发就完事了

TCP有拥塞控制、流量控制 UDP无

TCP首部开销大 默认20字节　UDP首部简单 默认8字节

TCP流式传输 UDP一个包一个包传送

> *TCP*为什么可靠传输

三次握手和四次挥手

序列号

确认应答

超时重传

流量控制

拥塞控制

> 用*udp* 实现*http*

UDP不可靠 但是基于UDP的QUIC协议实现可靠性传输 https3.0

> *tcp*粘包怎么解决？什么是粘包？

粘包指的是接收方接收时可能会将多个数据包粘在一起，导致接收方无法分辨每个数据包的边界

解决方法：

1、固定长度 填充空白（会导致浪费）

2、特殊字符作为边界

HTTP用回车符和换行符作为边界

3、自定义消息结构

包头和数据组成，其中包头包是固定大小的，而且包头里有一个字段来说明紧随其后的数据有多大。

TCP拥塞控制

发送窗口的值是swnd = min(cwnd, rwnd)，也就是拥塞窗口和接收窗口中的最小值。

> 慢启动

发送方每收到一个ACK 拥塞窗口cwnd大小加一

指数上涨（两倍） 直到cwnd大于等于 慢启动门限 启动避免算法

> 拥塞避免算法

进入拥塞避免后 每收到一个ACK 就增加 1/cwnd （线性增长） ++++++++++++++一个一个加

> 拥塞发生

如果上面两个算法 解决不了 发生重传

1.如果是超时重传

则cwnd初始化为1 门限除二 然后从1开始

2.如果是快速重传

收到3个重复ACK

cwnd除2 门限设置为除2后的cwnd 然后cwnd=门限+3

再收到重复ACK cwnd加1

收到新数据的ACK 则CWND重置为之前除2的门限 进入拥塞避免状态

NIO

内核空间和用户空间

内核空间：

- 内核空间是操作系统内核使用的内存区域。操作系统内核运行在特权模式下，它可以访问所有硬件资源和内存。内核空间有更高的权限，可以执行所有指令。
- 这个空间中的数据和代码可以访问整个计算机的硬件和其他系统资源（例如文件系统、网络、外设等）。
- 内核空间中一般存放操作系统内核、驱动程序、系统调用等关键代码。

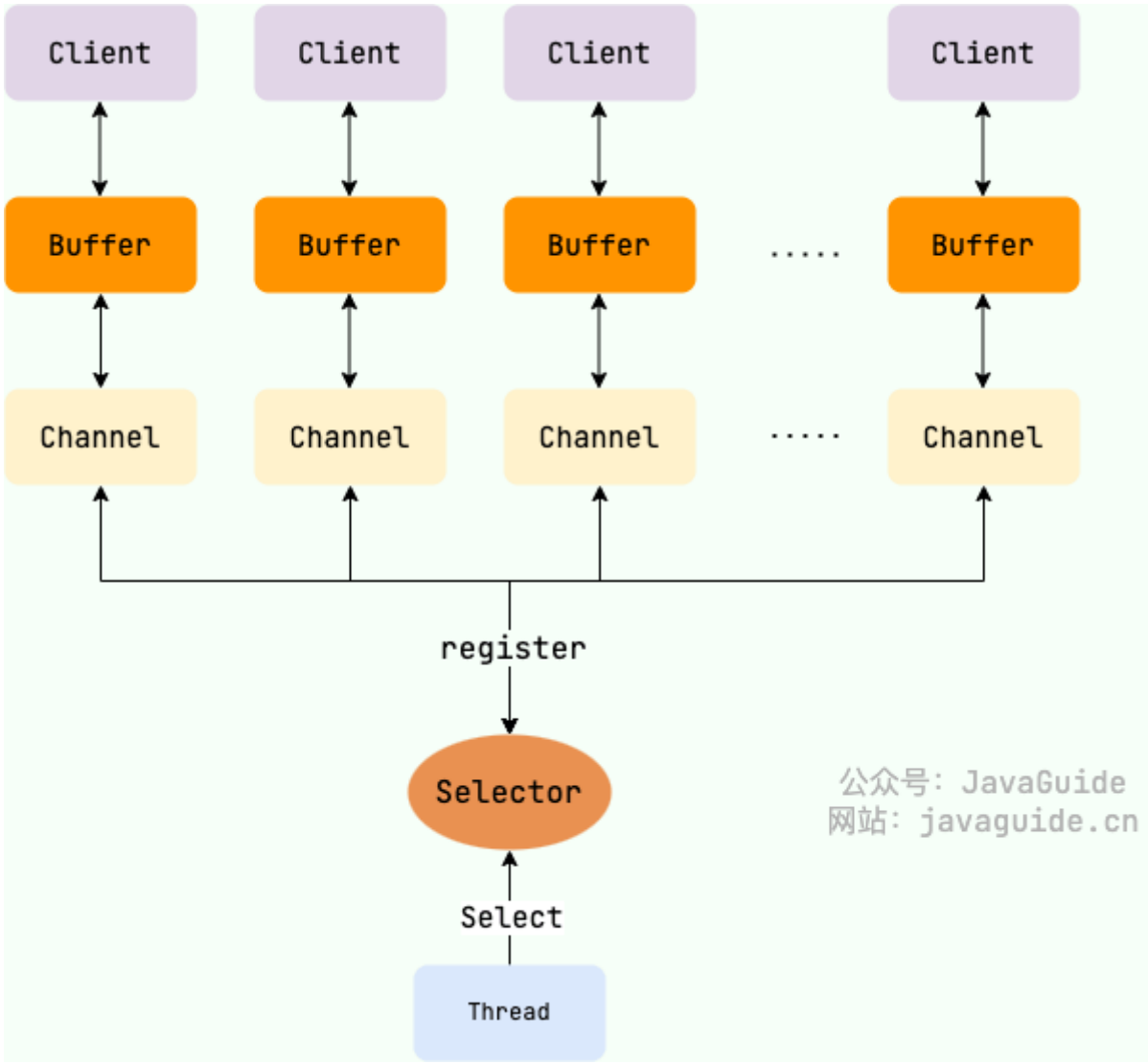
用户空间：

- 用户空间是应用程序运行的内存区域。在这个区域运行的程序受到操作系统的保护，不能直接访问硬件资源或内核空间。
- 用户空间中的程序只能通过系统调用或 API 请求内核空间进行资源操作。任何访问内核空间的请求必须经过操作系统的调度和控制。
- 用户空间中的数据和代码与操作系统和硬件是隔离的，目的是提供应用程序安全的运行环境。

最少最少都要两次**DMA**！ 一进一出

NIO三大件

- **Buffer**是数据的载体，用于存储数据。
- **Channel**是数据的传输通道，负责与Buffer交互。
- **Selector**是事件的管理者，监控多个Channel的状态，实现高效的多路复用。



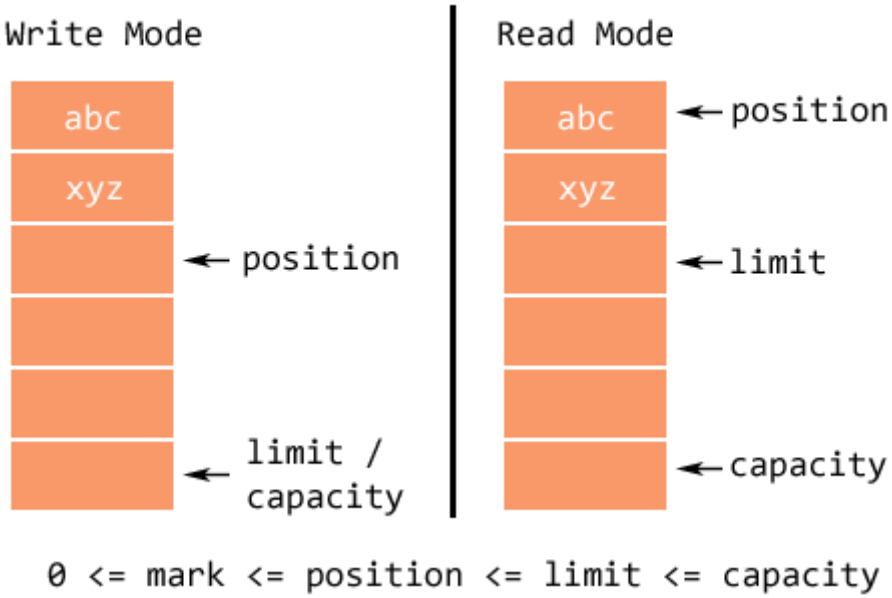
缓冲区（**Buffer**）：

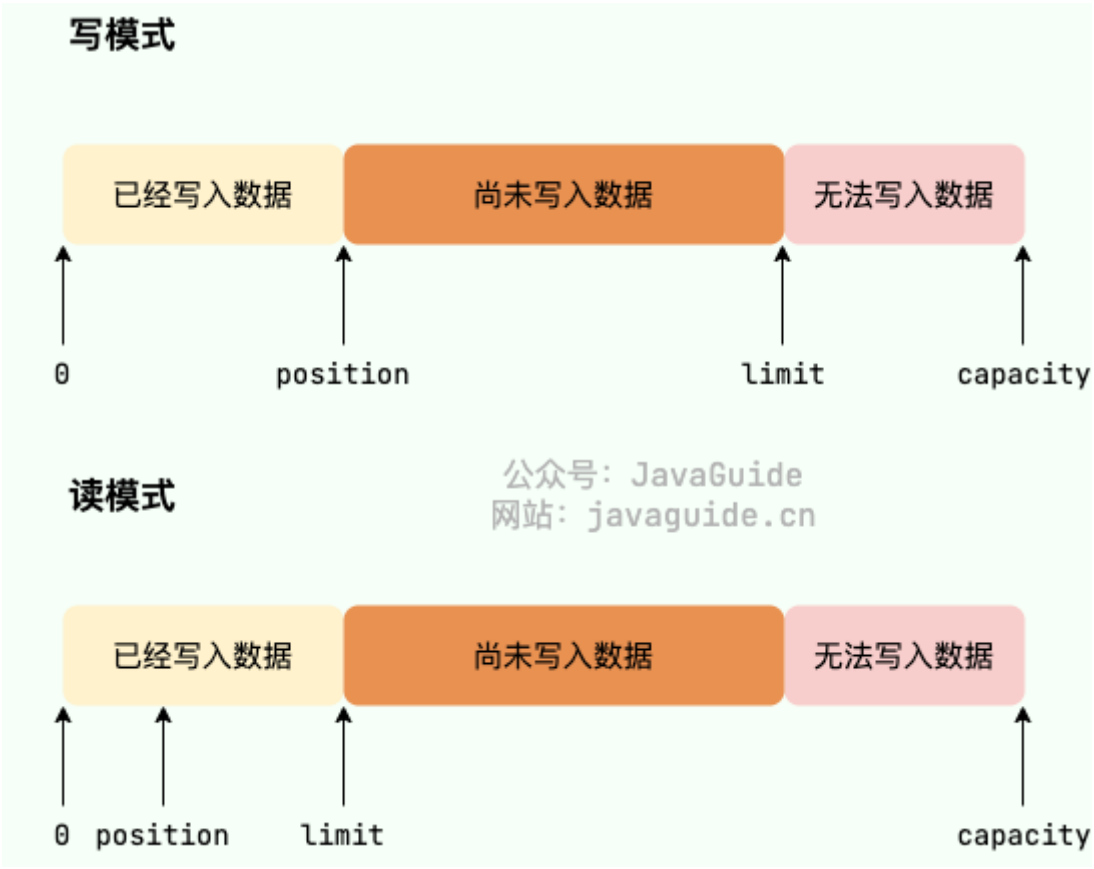
- 在传统 I/O 中，数据是通过流（Stream）来传输的。而在 NIO 中，数据是通过缓冲区（Buffer）来存储和传输的。
- 缓冲区是一个内存块，它包含一个可以读取和写入数据的数组。NIO 的所有 I/O 操作都涉及到缓冲区。

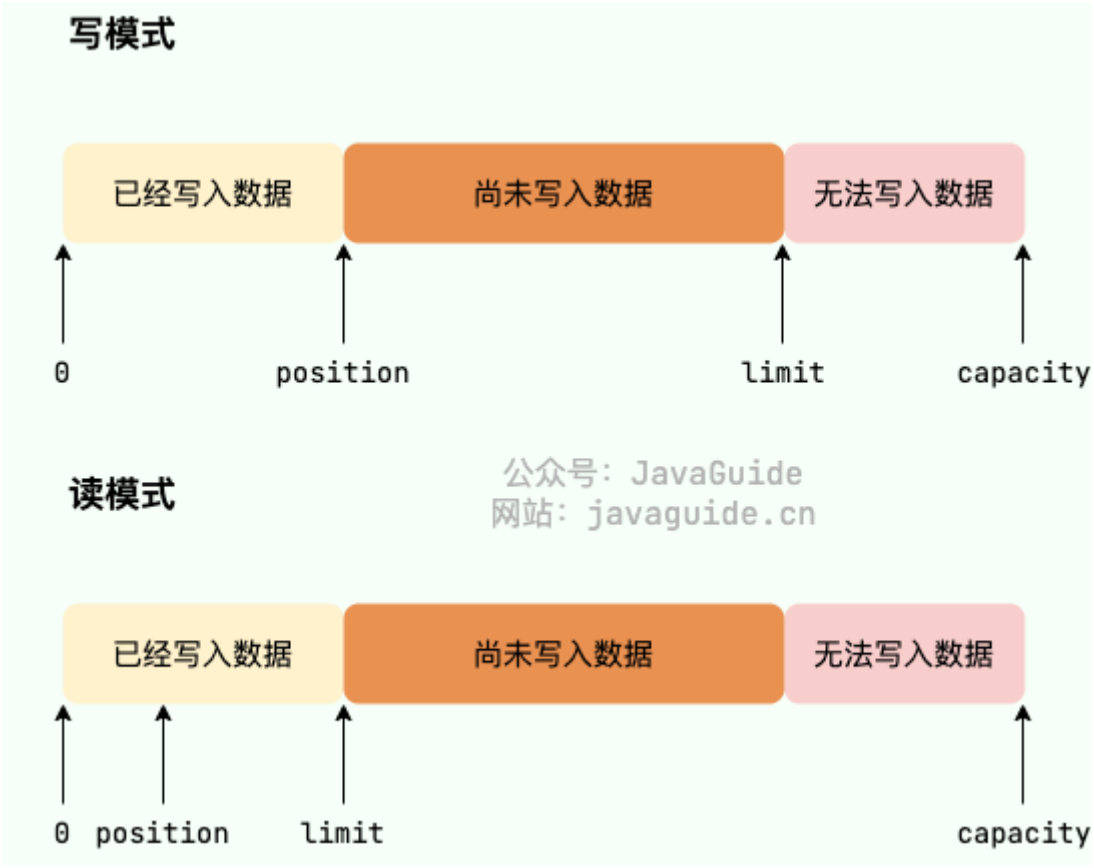
```
public abstract class Buffer {
    // Invariants: mark <= position <= limit <= capacity
    private int mark = -1;
    private int position = 0;
    private int limit;
    private int capacity;
}
```

1. 容量（`capacity`）：`Buffer`可以存储的最大数据量，`Buffer`创建时设置且不可改变；
2. 界限（`limit`）：`Buffer` 中可以读/写数据的边界。写模式下，`limit` 代表最多能写入的数据，一般等于 `capacity`（可以通过`limit(int newLimit)`方法设置）；读模式下，`limit` 等于 Buffer 中实际写入的数据大小。
3. 位置（`position`）：下一个可以被读写的数据的位置（索引）。从写操作模式到读操作模式切换的时候（flip），`position` 都会归零，这样就可以从头开始读写了。
4. 标记（`mark`）：`Buffer`允许将位置直接定位到该标记处，这是一个可选属性；

并且，上述变量满足如下的关系：**0 <= mark <= position <= limit <= capacity**







`Buffer` 对象不能通过 `new` 调用构造方法创建对象，只能通过静态方法实例化 `Buffer`。

Buffer 最核心的两个方法：

1. `get`：读取缓冲区的数据
2. `put`： 向缓冲区写入数据

这两个方法都是根据position来定位

除上述两个方法之外，其他的重要方法：

- `flip`： 将缓冲区从写模式切换到读模式，它会将 `limit` 的值设置为当前 `position` 的值，将 `position` 的值设置为 0。
- `clear`: 清空缓冲区，将缓冲区从读模式切换到写模式，并将 `position` 的值设置为 0，将 `limit` 的值设置为 `capacity` 的值。 注意 会换模式哦 不会真的清空 而是把limit和position重置

通道（**Channel**）：

- 通道类似于传统 I/O 中的流，但它可以双向传输数据（既可以读取数据，也可以写入数据）。全双工
- **FileChannel**、**SocketChannel** 和 **DatagramChannel** 等是常用的通道类型，它们分别用于文件 I/O、网络套接字 I/O 和数据报 I/O。.

其中，最常用的是以下几种类型的通道：

- `FileChannel`：文件访问通道；
- `SocketChannel`、`ServerSocketChannel`：TCP 通信通道；
- `DatagramChannel`：UDP 通信通道；

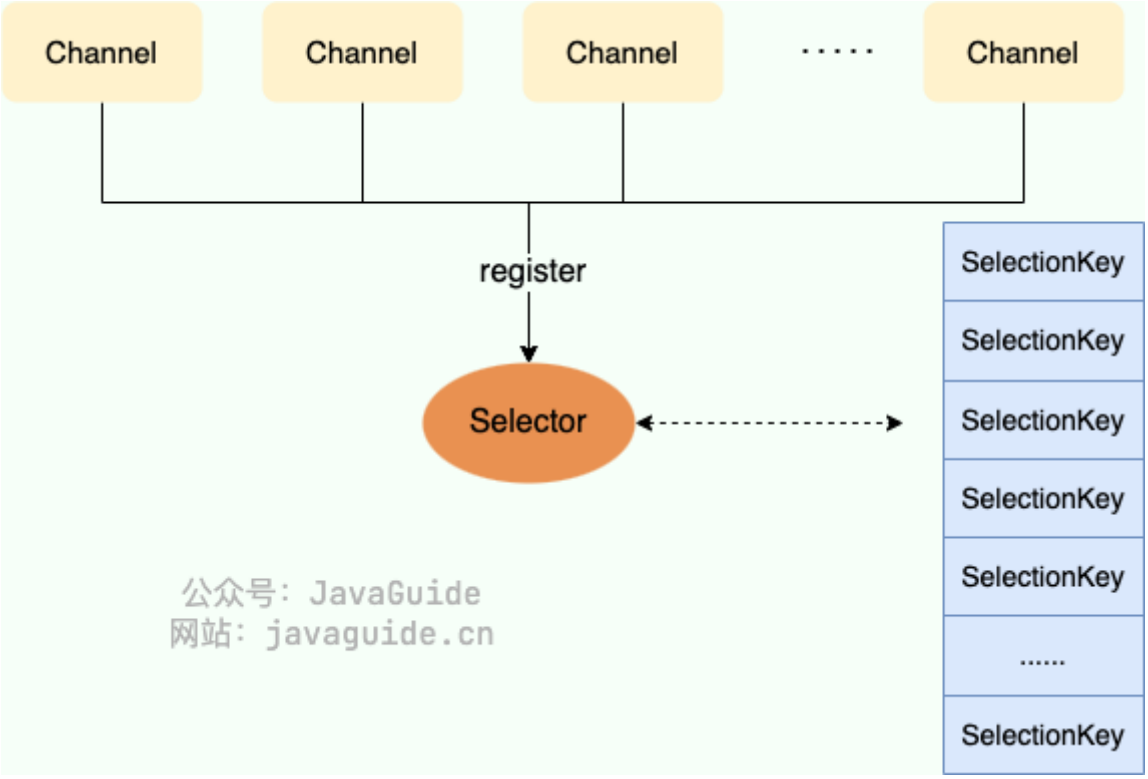
Channel 最核心的两个方法：

1. `read`： 读取数据并写入到 Buffer 中。
2. `write`： 将 Buffer 中的数据写入到 Channel 中。

选择器（**Selector**）：

- NIO 提供了一个 **Selector** 类，它允许单个线程监听多个通道（Channel）的事件。通过 **Selector**，我们可以在单线程中实现多路复用，处理多个连接或文件 I/O。
- 这就使得 NIO 特别适合于高并发的网络应用，可以高效地处理大量的并发连接。

某假设个 Channel 上面有新的 TCP 连接接入、读和写事件，这个 Channel 就处于就绪状态，会被 Selector 轮询出来。Selector 会将相关的 Channel 加入到就绪集合中。通过 SelectionKey 可以获取就绪 Channel 的集合，然后对这些就绪的 Channel 进行相应的 I/O 操作



Selector 可以监听以下四种事件类型：

1. `SelectionKey.OP_ACCEPT`：表示通道接受连接的事件，这通常用于 `ServerSocketChannel`。
2. `SelectionKey.OP_CONNECT`：表示通道完成连接的事件，这通常用于 `SocketChannel`。
3. `SelectionKey.OP_READ`：表示通道准备好进行读取的事件，即有数据可读。
4. `SelectionKey.OP_WRITE`：表示通道准备好进行写入的事件，即可以写入数据。

`Selector`是抽象类，可以通过调用此类的 `open()` 静态方法来创建 Selector 实例。Selector 可以同时监控多个 `SelectableChannel` 的 `IO` 状况，是非阻塞 `IO` 的核心。

一个 Selector 实例有三个 `SelectionKey` 集合：

1. 所有的 `SelectionKey` 集合：代表了注册在该 Selector 上的 `Channel`，这个集合可以通过 `keys()` 方法返回。
2. 被选择的 `SelectionKey` 集合：代表了所有可通过 `select()` 方法获取的、需要进行 `IO` 处理的 Channel，这个集合可以通过 `selectedKeys()` 返回。
3. 被取消的 `SelectionKey` 集合：代表了所有被取消注册关系的 `Channel`，在下一次执行 `select()` 方法时，这些 `Channel` 对应的 `SelectionKey` 会被彻底删除，

程序通常无须直接访问该集合，也没有暴露访问的方法。

简单案例：

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
while (keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if (key != null) {
        if (key.isAcceptable()) {
            // ServerSocketChannel 接收了一个新连接
        } else if (key.isConnectable()) {
            // 表示一个新连接建立
        } else if (key.isReadable()) {
            // Channel 有准备好的数据，可以读取
        } else if (key.isWritable()) {
            // Channel 有空闲的 Buffer，可以写入数据
        }
    }
    keyIterator.remove();
}
```

简单讲：新建channel和新建一个selector 然后将channel注册到selector （包括想要监听的事件） 用循环来轮值获取准备好的channel，把准备好的channel塞进set里面 用set的迭代器循环获取 每个channel对应的SelectionKey 然后根据key来做对应的操作 操作完当前的key后 删除迭代器这个key

番外：

`SelectionKey` 是与通道相关的事件标识符，每个注册到 `Selector` 上的通道都会对应一个 `SelectionKey`。`SelectionKey` 保存了有关通道、事件类型和其它元数据的信息。

`isAcceptable()` 和其他 `is` 方法（如 `isReadable()`, `isWritable()`）是基于 `SelectionKey` 中的 就绪事件 来判断的

`SelectionKey` 对象包含以下重要信息：

- `channel`：通道（如 `ServerSocketChannel` 或 `SocketChannel`）。
- `interestOps`：兴趣操作集（表示你希望监听的事件，如 `OP_ACCEPT`、`OP_READ`、`OP_WRITE`）。
- `readyOps`：已经就绪的操作集（表示当前可以进行的操作，如 `OP_ACCEPT` 表示连接请求可接受）。

`readyOps` 只能是 `interestOps` 的子集。即，`readyOps` 中的事件必须是你在 `interestOps` 中注册过的事件。

是的，如果 `interestOps` 与 `readyOps` 不符，那就意味着当前没有准备好的事件可以处理。具体来说：

- `interestOps` 定义了你希望 `Selector` 监听哪些操作（如 `OP_READ`、`OP_WRITE`、`OP_ACCEPT` 等），但并不意味着这些操作总是会发生。
- `readyOps` 则表示当前 `Selector` 返回的、已准备好的事件。只有在 `readyOps` 中包含的操作才会被处理。

处理逻辑：

1. 如果 `interestOps` 和 `readyOps` 不匹配：
    - 如果你在 `interestOps` 中注册了某个事件（比如 `OP_READ`），但 `readyOps` 中没有该事件，说明该通道当前并不准备好进行该操作。
    - 这种情况下，`Selector` 不会返回该通道，或者即使返回了，也不会执行任何操作。你会跳过该通道，继续监听其他通道的事件。
2. 继续等待其他通道：
    - `Selector.select()` 或 `selectNow()` 会一直等待，直到有通道的事件准备好。这时，它会返回包含准备好的事件的 `SelectionKey` 集合，你可以通过 `key.isReadable()`、`key.isWritable()` 等方法检查具体的操作。
    - 如果当前没有通道准备好某个事件，`Selector` 会返回 `readyOps` 为空的情况，或者只包含部分你关注的事件。此时，程序会继续等待直到有其他事件可处理。

非阻塞 **I/O**：

- 传统 I/O 通常是阻塞式的，即一个线程读取数据时，如果没有数据可读，它就会被阻塞，直到有数据到达。
- NIO 通过非阻塞 I/O（non-blocking I/O）来避免这种阻塞情况。一个线程可以检查通道是否有数据可用，如果没有数据，它可以继续执行其他任务，而不是被阻塞在 I/O 操作上。

零拷贝

NIO做到了零拷贝

传统的 I/O 操作中，数据从硬盘、网络等输入输出设备传输到内存时，通常需要经历多个步骤：

1. 从磁盘读取数据到内核缓冲区。
2. 从内核缓冲区复制到用户空间缓冲区（如果是网络 I/O，则是通过套接字进行数据传输）。
3. 从用户空间缓冲区写回到网络或磁盘。

这些步骤都涉及到多次内存拷贝操作，造成了不必要的性能损失。尤其是当数据量很大时，这些内存拷贝会显著降低系统性能

零拷贝（Zero-Copy）是一种优化技术，旨在减少数据从一个地方复制到另一个地方时的内存拷贝操作。其核心思想是尽可能避免在数据传输过程中将数据从内核空间复制到用户空间，再从用户空间复制回内核空间，从而减少 CPU 和内存的开销。

数据可以直接从 内核空间 传输到 内核空间（例如从磁盘到网络，或者从磁盘到内存），或者从 内核空间 传输到 用户空间，而无需通过中间的内存复制。

NIO 引入了 通道（**Channel**） 和 缓冲区（**Buffer**） 的概念，其中 **FileChannel** 提供了一种 零拷贝 的方式来减少不必要的数据复制。

可以在文件和网络数据传输中直接将数据从一个通道传输到另一个通道

NIO 中的 `FileChannel` 类提供了两种非常重要的方法，能够实现零拷贝：

1. **transferTo()**
2. **transferFrom()**

这两个方法可以直接在操作系统内核中进行数据传输，避免了将数据从内核缓冲区拷贝到用户缓冲区的过程。因此，减少了数据传输过程中的额外内存复制操作，从而实现了零拷贝。

零拷贝的常见实现技术有： `mmap+write`、`sendfile`和 `sendfile + DMA gather copy` 。

> *IO多路复用*

IO多路复用（也是NIO） 调用select 监听多个sockets 有socket准备好就返回readable 此过程用户进程阻塞 第二阶段 非阻塞 调用recvfrom从就绪socket读取数据 内核拷贝数据到用户数据 也是阻塞

多路复用：通道状态放在内核空间 内核空间查channel事件 而不是用户空间来查(不然会反复切换) 用户空间把要查的通道传给内核空间 让内核空间查完 返回状态到用户空间 切换用户和内核即阻塞 阻塞一次就能查到状态了

select poll epoll

> *select*和*poll*区别

`select` 会将多个 I/O 通道（如 socket）和它们对应的事件（如读、写、异常等）传给操作系统，由内核对这些通道进行监视。当某个通道准备好事件时，`select` 会通知应用程序可以进行相关的 I/O 操作。

缺点：

- 文件描述符限制：`select` 有文件描述符的上限，通常是 1024（这个数字在一些系统中可以调整，但仍然有限）。如果连接数超过这个限制，`select` 就无法再监视更多的文件描述符。
- 效率低下：每次调用 `select` 都需要遍历所有的文件描述符，检查它们是否有事件发生，导致其在大量连接时性能较差。特别是当没有事件发生时，`select` 会浪费大量的 CPU 资源。
- 每次调用都传递整个文件描述符集合：每次调用 `select` 时，需要传递整个文件描述符集合，即使只是检查一个通道，也必须重新传递所有通道，效率低。

`poll` 使用动态分配的数组来存储文件描述符，可以监控更多的文件描述符。 事件标志更多。

select 函数使用 fd_set 类型的描述符集合（固定大小的位图）来传递需要监视的文件描述符列表，而 poll 函数使用 struct pollfd 数组类型（数组大小是动态的）来指定需要监听的文件描述符及其感兴趣的事件。

select 函数每次调用时都需要将待检测的文件描述符集合拷贝到内核中去，而 poll 函数则是通过参数传递文件描述符集合，不需要对文件描述符集合进行操作

poll使用的是pollfd，是一种链式的结构，没有最大文件描述符限制。

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

*fds指向 `pollfd` 结构体数组的指针。

`fd_set` 是用户空间的位图，内核无法直接访问 `poll` 的 `pollfd` 数组是通过指针传递的，内核可以直接访问用户空间的内存。 **select**没有设计成指针 是因为历史遗留罢了

> *epoll*

不用再遍历了

epoll_create 内核空间里开启红黑树和双向链表（就绪队列）

epoll_ctl `epoll` 实例中添加、修改或删除需要监控的文件描述符 将fd描述符放在红黑树里 包括监听事件

epoll_wait 遍历链表 拿可连接的fd 当某个被监控的文件描述符上发生了事件（如可读、可写等），内核会将该文件描述符添加到就绪队列中。

有数据---》触发可读--》到红黑树里面查----》匹配了就放到就绪队列 （红黑树不再存在该节点）

服务限流

固定窗口

其实就是时间窗口，其原理是将时间划分为固定大小的窗口，在每个窗口内限制请求的数量或速率，即固定窗口计数器算法规定了系统单位时间处理的请求数量。

超过就归0

缺点：

- 限流不够平滑。例如，我们限制某个接口每分钟只能访问 30 次，假设前 30 秒就有 30 个请求到达的话，那后续 30 秒将无法处理请求，这是不可取的，用户体验极差！
- 无法保证限流速率，因而无法应对突然激增的流量。例如，我们限制某个接口 1 分钟只能访问 1000 次，该接口的 QPS 为 500，前 55s 这个接口 1 个请求没有接收，后 1s 突然接收了 1000 个请求。然后，在当前场景下，这 1000 个请求在 1s 内是没办法被处理的，系统直接就被瞬时的大量请求给击垮了。

滑动窗口

滑动窗口计数器算法 算的上是固定窗口计数器算法的升级版，限流的颗粒度更小。

滑动窗口计数器算法相比于固定窗口计数器算法的优化在于：它把时间以一定比例分片 。

例如我们的接口限流每分钟处理 60 个请求，我们可以把 1 分钟分为 60 个窗口。每隔 1 秒移动一次，每个窗口一秒只能处理不大于 `60(请求数)/60（窗口数）` 的请求， 如果当前窗口的请求计数总和超过了限制的数量的话就不再处理其他请求。

很显然， 当滑动窗口的格子划分的越多，滑动窗口的滚动就越平滑，限流的统计就会越精确。

缺点：

- 与固定窗口计数器算法类似，滑动窗口计数器算法依然存在限流不够平滑的问题。
- 相比较于固定窗口计数器算法，滑动窗口计数器算法实现和理解起来更复杂一些。

漏桶算法

我们可以把发请求的动作比作成注水到桶中，我们处理请求的过程可以比喻为漏桶漏水。我们往桶中以任意速率流入水，以一定速率流出水。当水超过桶流量则丢弃，因为桶容量是不变的，保证了整体的速率。

如果想要实现这个算法的话也很简单，准备一个队列用来保存请求，然后我们定期从队列中拿请求来执行就好了（和消息队列削峰/限流的思想是一样的）。

缺点：

- 无法应对突然激增的流量，因为只能以固定的速率处理请求，对系统资源利用不够友好。
- 桶流入水（发请求）的速率如果一直大于桶流出水（处理请求）的速率的话，那么桶会一直是满的，一部分新的请求会被丢弃，导致服务质量下降。

实际业务场景中，基本不会使用漏桶算法。

令牌桶

不过现在桶里装的是令牌了，请求在被处理之前需要拿到一个令牌，请求处理完毕之后将这个令牌丢弃（删除）。我们根据限流大小，按照一定的速率往桶里添加令牌。如果桶装满了，就不能继续往里面继续添加令牌了。

缺点：

- 如果令牌产生速率和桶的容量设置不合理，可能会出现问题比如大量的请求被丢弃、系统过载。
- 相比于其他限流算法，实现和理解起来更复杂一些。

其他

针对 IP 进行限流是目前比较常用的一个方案。不过，实际应用中需要注意用户真实 IP 地址的正确获取。常用的真实 IP 获取方法有 X-Forwarded-For 和 TCP Options 字段承载真实源 IP 信息。虽然 X-Forwarded-For 字段可能会被伪造，但因为其实现简单方便，很多项目还是直接用的这种方法。

除了我上面介绍到的限流对象之外，还有一些其他较为复杂的限流对象策略，比如阿里的 Sentinel 还支持 基于调用关系的限流（包括基于调用方限流、基于调用链入口限流、关联流量限流等）以及更细维度的 热点参数限流（实时的统计热点参数并针对热点参数的资源调用进行流量控制）

单机限流针对的是单体架构应用。

单机限流可以直接使用 Google Guava 自带的限流工具类 `RateLimiter` 。 `RateLimiter` 基于令牌桶算法，可以应对突发流量。

> *Guava 地址：https://github.com/google/guava*

除了最基本的令牌桶算法(平滑突发限流)实现之外，Guava 的`RateLimiter`还提供了 平滑预热限流 的算法实现。

平滑突发限流就是按照指定的速率放令牌到桶里，而平滑预热限流会有一段预热时间，预热时间之内，速率会逐渐提升到配置的速率。

```
// 1s 放 5 个令牌到桶里也就是 0.2s 放 1个令牌到桶里
        // 预热时间为3s,也就说刚开始的 3s 内发牌速率会逐渐提升到 0.2s 放 1 个令牌到桶里
        RateLimiter rateLimiter = RateLimiter.create(5, 3, TimeUnit.SECONDS);
```

Resilience4j 是一个轻量级的容错组件，其灵感来自于 Hystrix。自Netflix 宣布不再积极开发 Hystrix 之后，Spring 官方和 Netflix 都更推荐使用 Resilience4j 来做限流熔断。

> *Resilience4j 地址: https://github.com/resilience4j/resilience4j*

分布式限流常见的方案：

- 借助中间件限流：可以借助 Sentinel 或者使用 Redis 来自己实现对应的限流逻辑。
- 网关层限流：比较常用的一种方案，直接在网关层把限流给安排上了。不过，通常网关层限流通常也需要借助到中间件/框架。就比如 Spring Cloud Gateway 的分布式限流实现`RedisRateLimiter`就是基于 Redis+Lua 来实现的，再比如 Spring Cloud Gateway 还可以整合 Sentinel 来做限流。

为什么建议 **Redis+Lua** 的方式？ 主要有两点原因：

- 减少了网络开销：我们可以利用 Lua 脚本来批量执行多条 Redis 命令，这些 Redis 命令会被提交到 Redis 服务器一次性执行完成，大幅减小了网络开销。
- 原子性：一段 Lua 脚本可以视作一条命令执行，一段 Lua 脚本执行过程中不会有其他脚本或 Redis 命令同时执行，保证了操作不会被其他指令插入或打扰。

Redisson 中的 `RRateLimiter` 来实现分布式限流，其底层实现就是基于 Lua 代码+令牌桶算法。

Redisson 是一个开源的 Java 语言 Redis 客户端，提供了很多开箱即用的功能，比如 Java 中常用的数据结构实现、分布式锁、延迟队列等等。并且，Redisson 还支持 Redis 单机、Redis Sentinel、Redis Cluster 等多种部署架构。

`RRateLimiter` 的使用方式非常简单。我们首先需要获取一个`RRateLimiter`对象，直接通过 Redisson 客户端获取即可。然后，设置限流规则就好。

```
// 创建一个 Redisson 客户端实例
RedissonClient redissonClient = Redisson.create();
// 获取一个名为 "javaguide.limiter" 的限流器对象
RRateLimiter rateLimiter =
redissonClient.getRateLimiter("javaguide.limiter");
// 尝试设置限流器的速率为每小时 100 次
// RateType 有两种，OVERALL是全局限流,ER_CLIENT是单Client限流（可以认为就是单机限流）
rateLimiter.trySetRate(RateType.OVERALL, 100, 1,
RateIntervalUnit.HOURS);

接下来我们调用acquire()方法或tryAcquire()方法即可获取许可。

    // 获取一个许可，如果超过限流器的速率则会等待
// acquire()是同步方法，对应的异步方法：acquireAsync()
rateLimiter.acquire(1);
// 尝试在 5 秒内获取一个许可，如果成功则返回 true，否则返回 false
// tryAcquire()是同步方法，对应的异步方法：tryAcquireAsync()
boolean res = rateLimiter.tryAcquire(1, 5, TimeUnit.SECONDS);
```