# 计算机网络部分

## 从输入URL到页面展示到底发生了什么？

第一步是**用户输入URL**，并按下回车。此时，浏览器开始**解析URL**，将其分解为**协议**（例如HTTP或HTTPS）、**域名**、**路径**（例如/home）等部分。

第二步是进行**DNS解析**。浏览器通过DNS解析域名，**查找对应的IP地址**。若该域名的IP地址**已被缓存**，浏览器会**直接使用缓存的IP地址**；若**未缓存**，则浏览器会向DNS服务器**发送请求，获取IP地址**。

第三步是通过三次握手**建立TCP连接**。获取到IP地址后，浏览器通过**TCP/IP协议**与目标服务器**建立连接**。在HTTPS请求中，浏览器还会通过**SSL协议进行加密**，确保通信的安全性。

第四步是**客户端发送HTTP请求**。TCP连接建立后，浏览器会发送HTTP请求到服务器。请求包括**请求行、请求头和请求体**。**请求行包含请求方法（如GET、POST）、请求的路径和HTTP版本**。请求头包括目标域名、浏览器身份标识、可以处理的内容类型等信息。**请求体则用于发送数据（如POST表单提交数据）**。

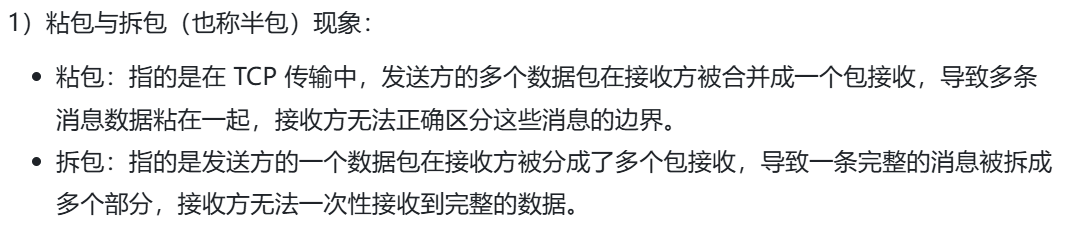
第五步是**服务器处理请求**。服务器接收到请求后，首先会解析请求头并**根据请求的URL路径**选择合适的处理方式。

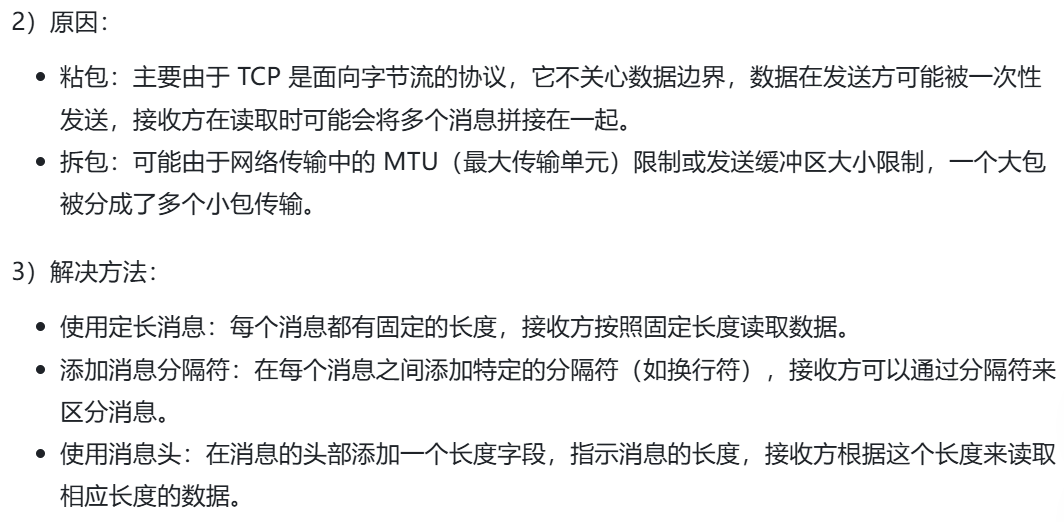
第六步是**服务器发送HTTP响应**。服务器处理完请求后，构造HTTP响应并返回给浏览器。**响应包含响应行、响应头和响应体**。**响应行指示处理结果（如200 OK、404 NotFound）**，响应头包含响应体的数据类型、响应体的长度等信息，**响应体则是实际的页面内容（HTML、CSS、JavaScript、图片等）**。

第七步是**浏览器解析和渲染页面**。浏览器接收到响应后，开始解析响应体中的HTML内容。

第八步是通过四次挥手**关闭TCP连接**。

## TCP的粘包和拆包？





## 常见的HTTP状态码？





## TCP和UDP的区别？



## GET和POST的区别

GET和POST的5点主要区别：

第一个是数据传输方式的区别：GET请求**通过URL传递数据**，数据被附加在URL后面，以键值对的形式传输。而POST请求将数据**放在请求体中传递**，数据不会显示在URL中。

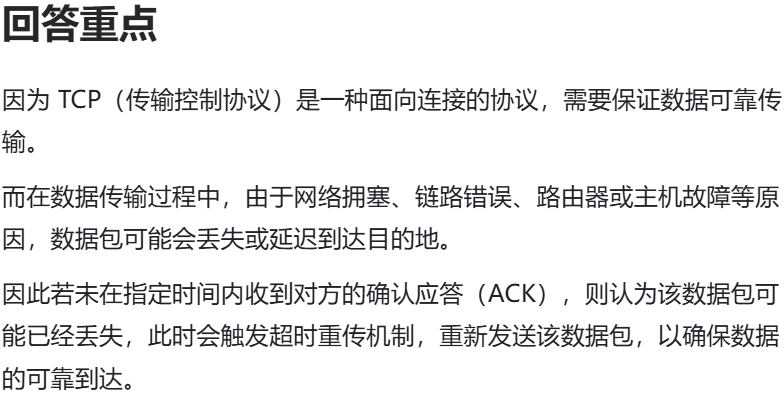
第二个是数据大小限制的区别：GET**请求的数据大小有限制**，通常为2至8KB。而POST请求的数据没有固定限制，**可以传输大量数据**，理论上只有Web服务器的配置限制。

第三个是安全性的区别：GET请求的数据通过URL传递，**数据泄露**给第三方，安全性较差。而POST请求的数据**保存在请求体中**，不会显示在URL中，**相对更安全**一些。

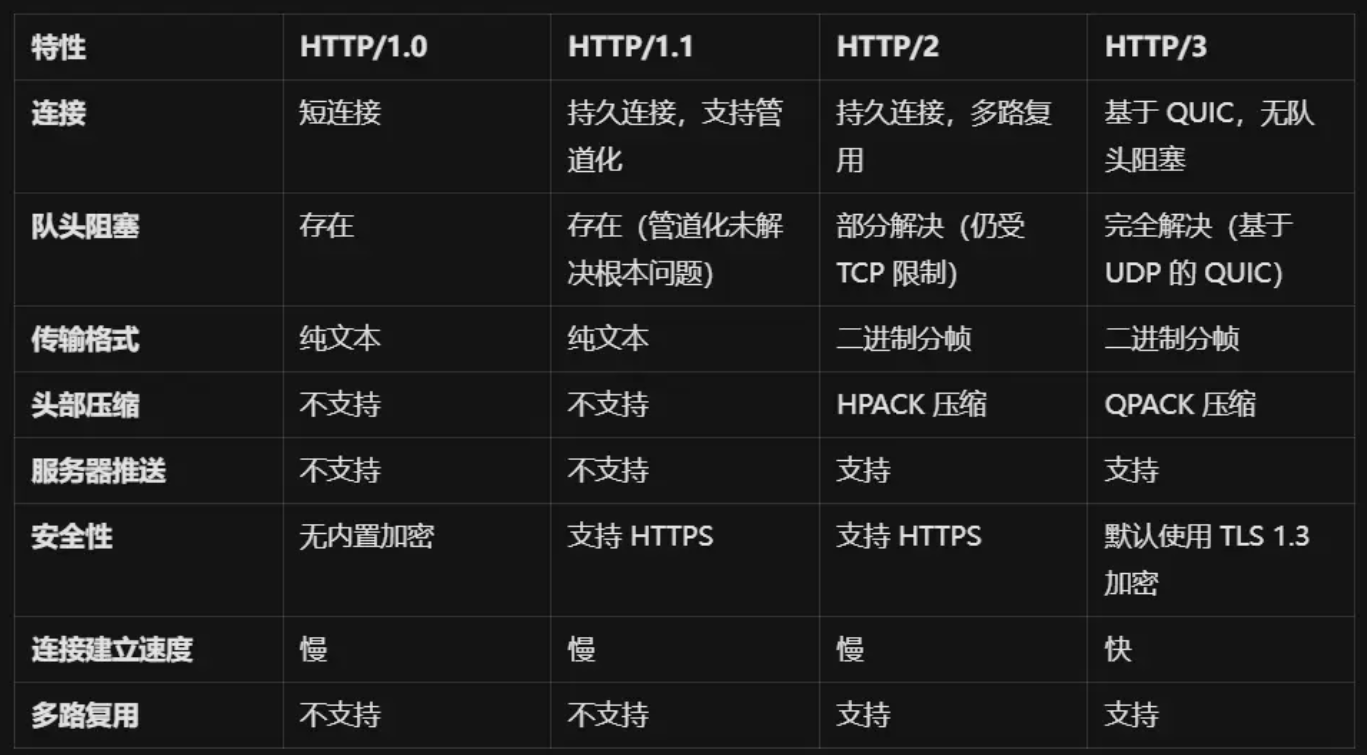
第四个是使用场景的区别：GET请求**通常用于获取资源或数据，适用于查询操作**。而POST请求通常用于**提交数据或进行数据更改**操作，适用于表单提交、用户注册、登录、文件上传等场景。

第五个是**幂等性**和缓存的区别：GET请求是幂等的，即同样的请求多次发送，服务器的响应不会发生变化，且可以被**缓存**。而POST请求是非幂等的，每次请求都会产生变化，因此不能缓存。

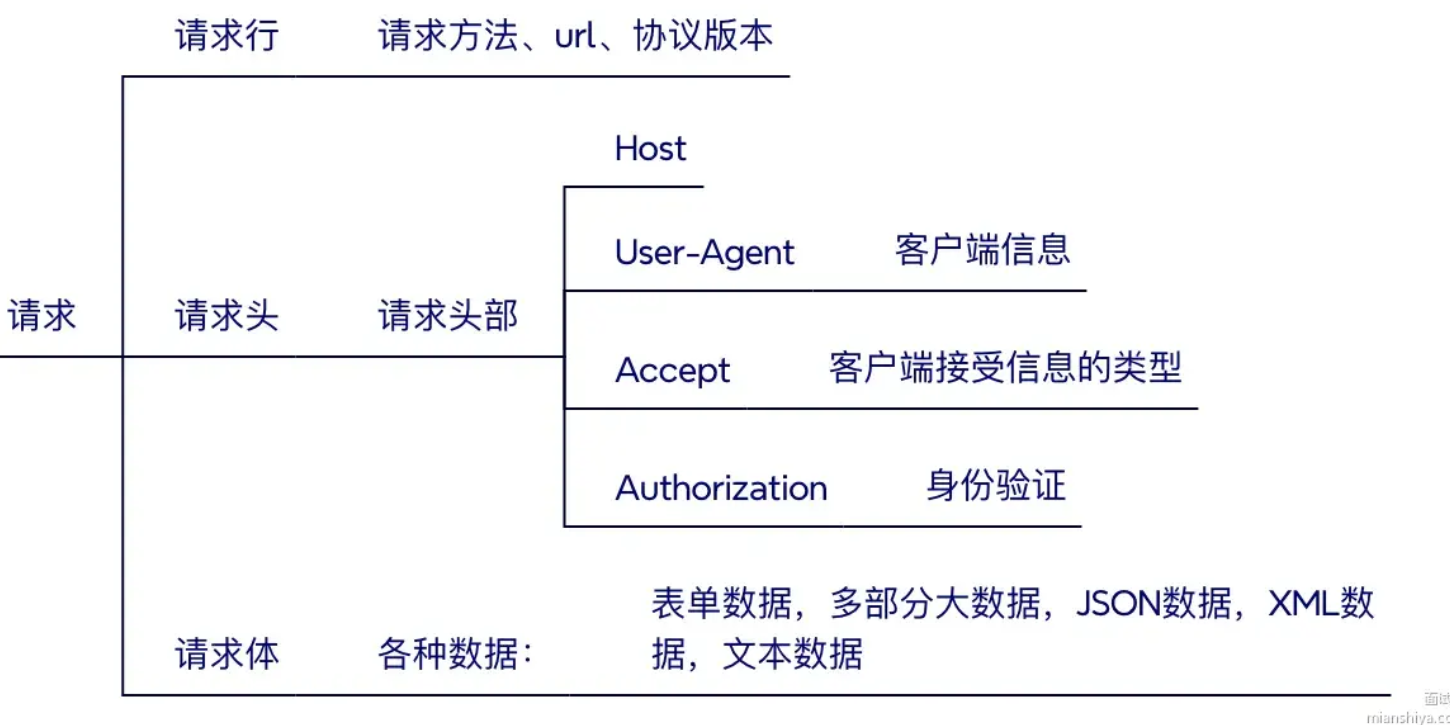
## TCP超时重传机制为了解决什么问题？



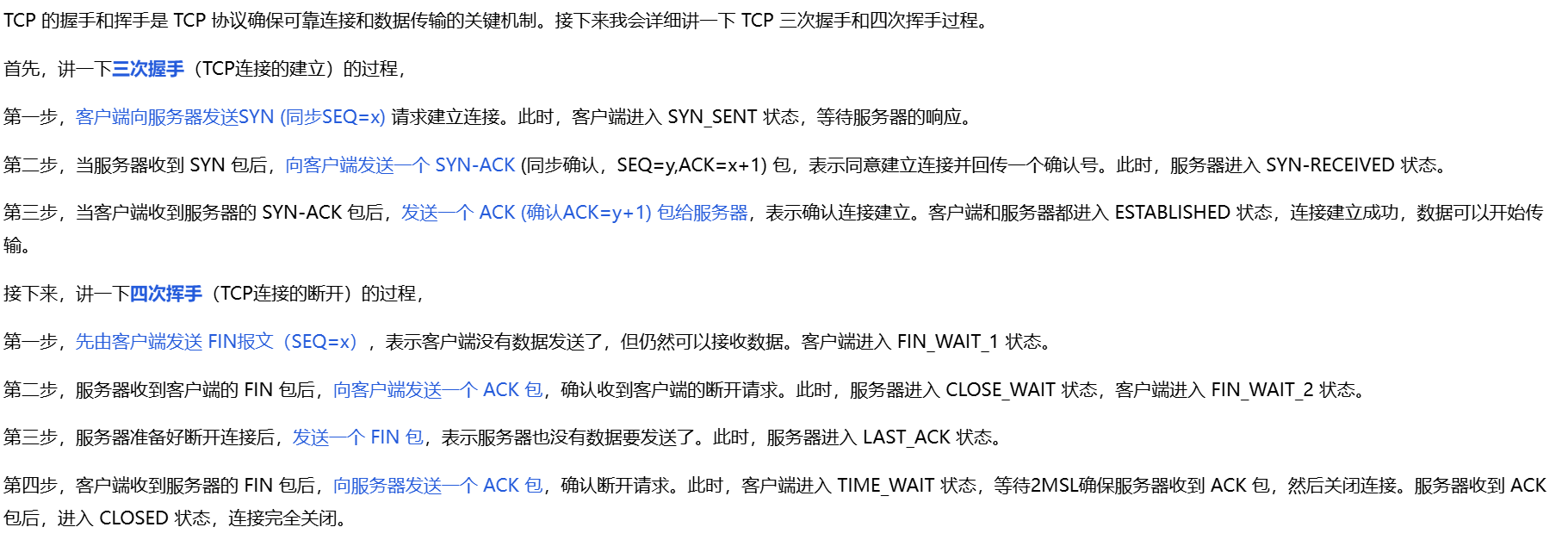
## HTTP1.0、2.0和3.0的区别

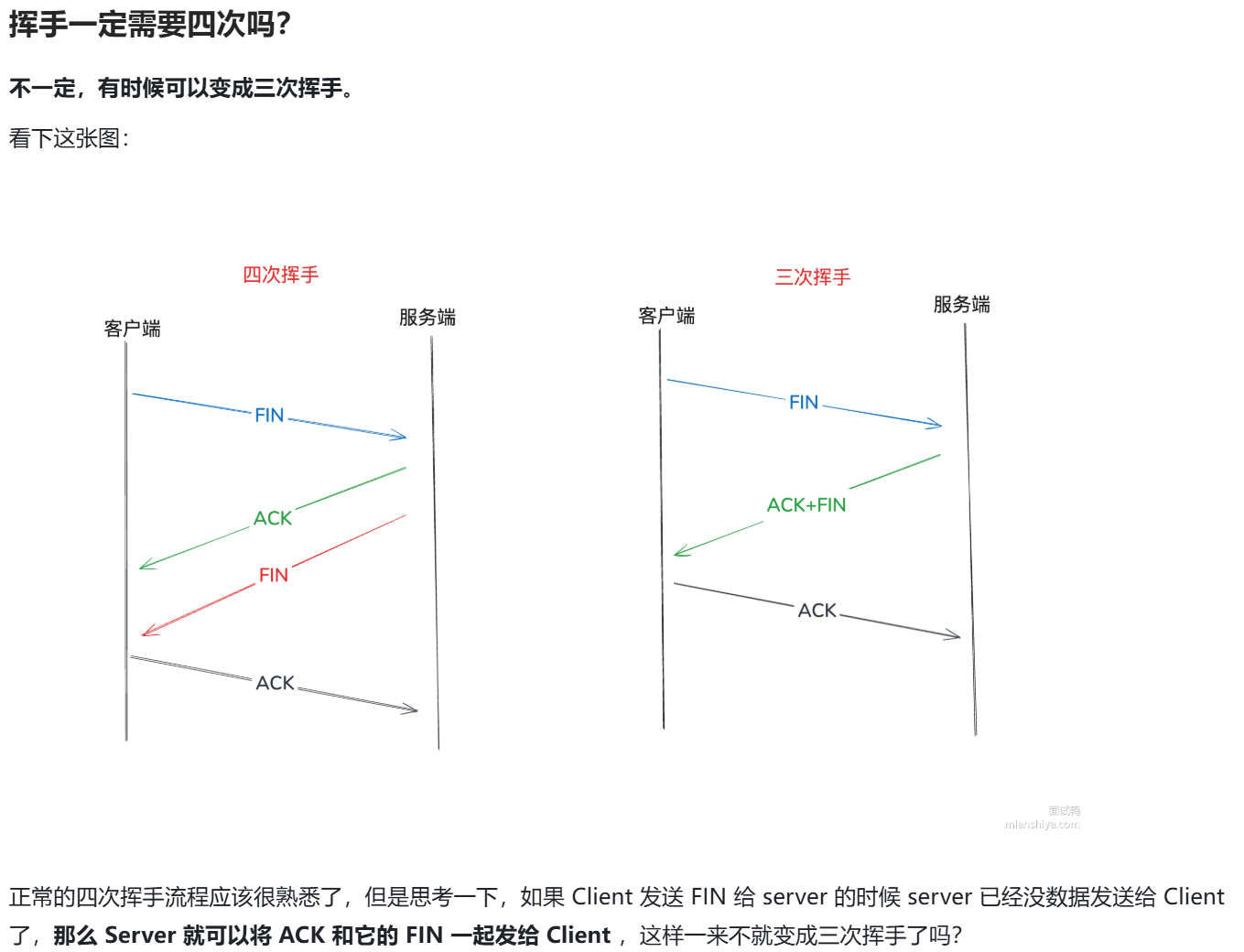


## 服务器如何解析HTTP请求的数据



## 三次握手和四次挥手？





## 为什么要等2MSL？为什么是等待2MSL？

（1）确保客户端最后的ACK能够被成功接收。因为如果这个ACK丟失了，服务器没有收到确认包，会重新发送FIN报文，而MSL是TCP报文在网络中可以存活的最大时间，服务器重发FIN，客户端收到之后重发ACK，这一来一回就需要2MSL的时间。

（2）防止旧的报文干扰新的连接。TCP连接关闭之后，可能会有一些延迟的或者已经失效的报文还在网络中传输，如果我们立即用相同的IP地址和端口建立新的连接，可能会受到这些旧的报文的干扰。

## TCP实现可靠传输的原理

首先是**连接管理机制**，TCP通过**三次握手建立连接**，确保双方通信正常，并用**四次挥手终止连接**，防止数据残留或资源浪费。

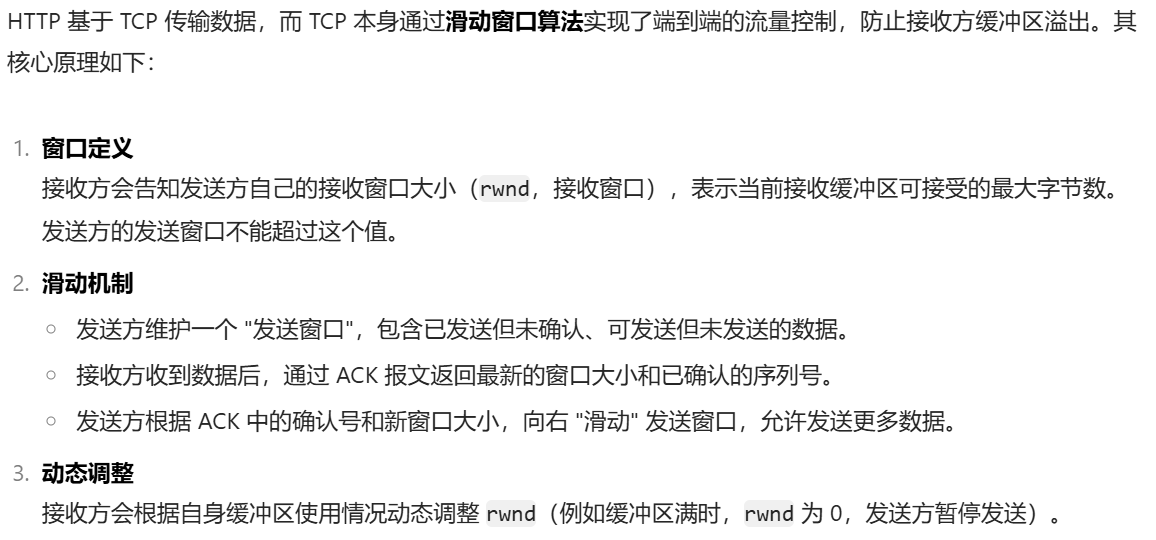
第二个是数据分块与序号标识机制，发送端将**数据分割为合适大小的报文段**，每个段分配**唯一序号**，**标识数据的顺序**；接收端**通过序号重组乱序到达的段**，确保数据完整性。

第三个是**确认应答与超时重传机制**，接收端对每个接收到的段返回确认应答，发送端如果超时未收到ACK就会触发超时重传，解决数据丢失问题。

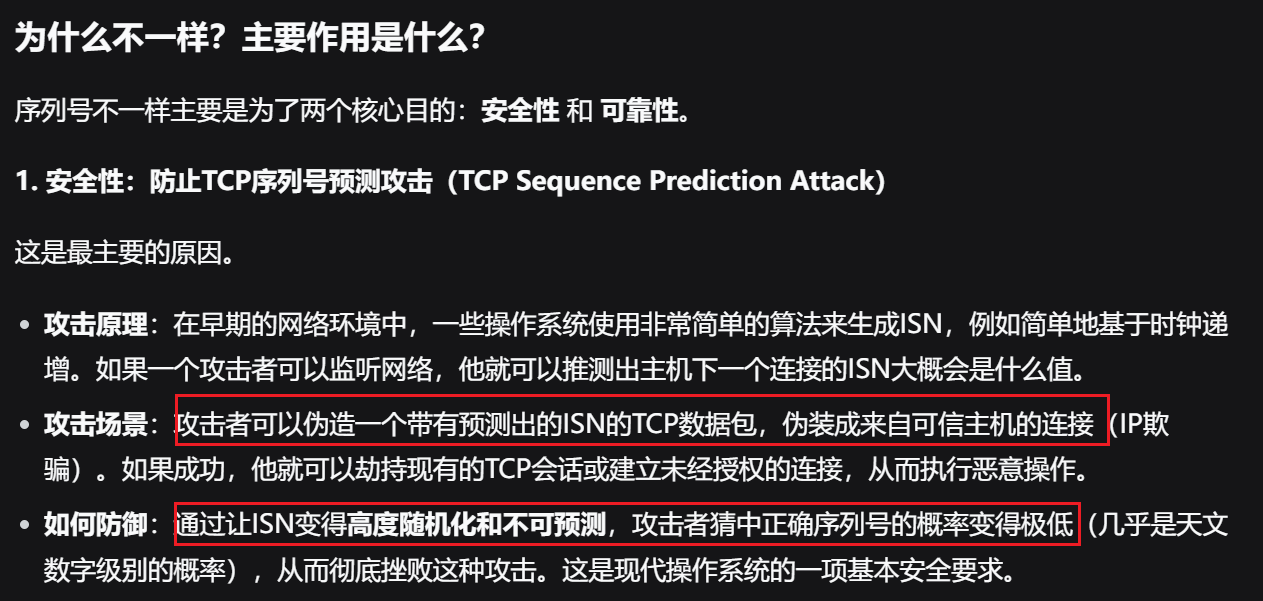
第四个是**流量控制**机制，接收端**通过窗口大小告知发送端可接收的数据量**，**避免缓冲区溢出**。滑动窗口机制允许连续发送多个段，提升传输效率。

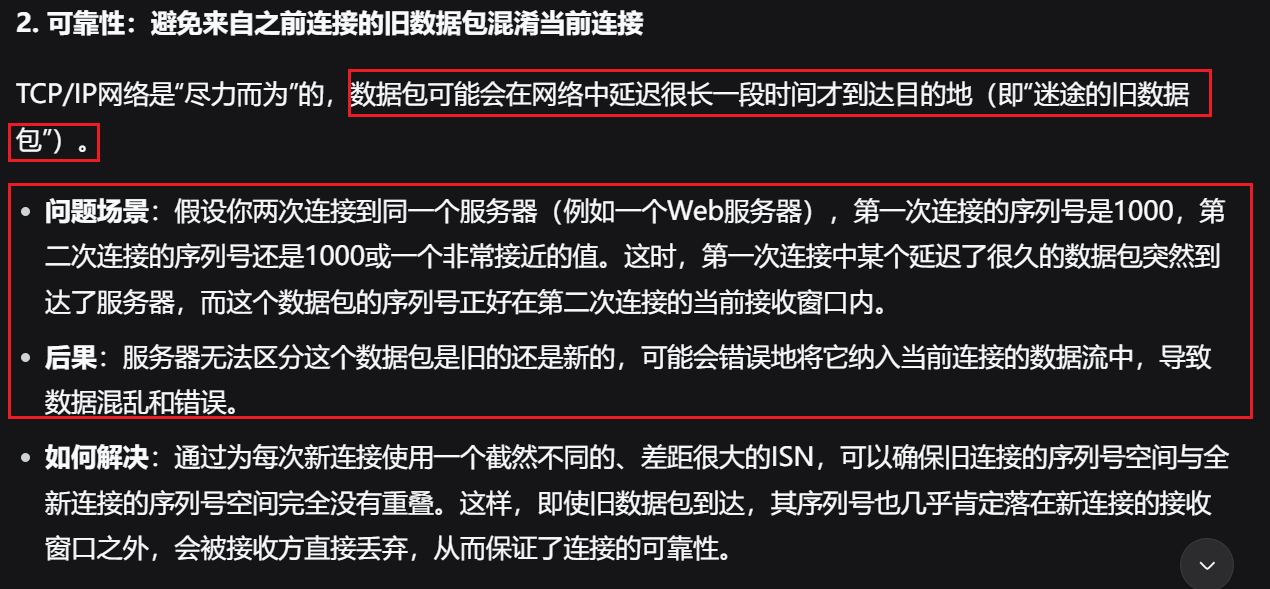
第五个是**拥塞控制**机制，能够根据网络负载情况**动态调整发送速率**，防止网络瘫痪。

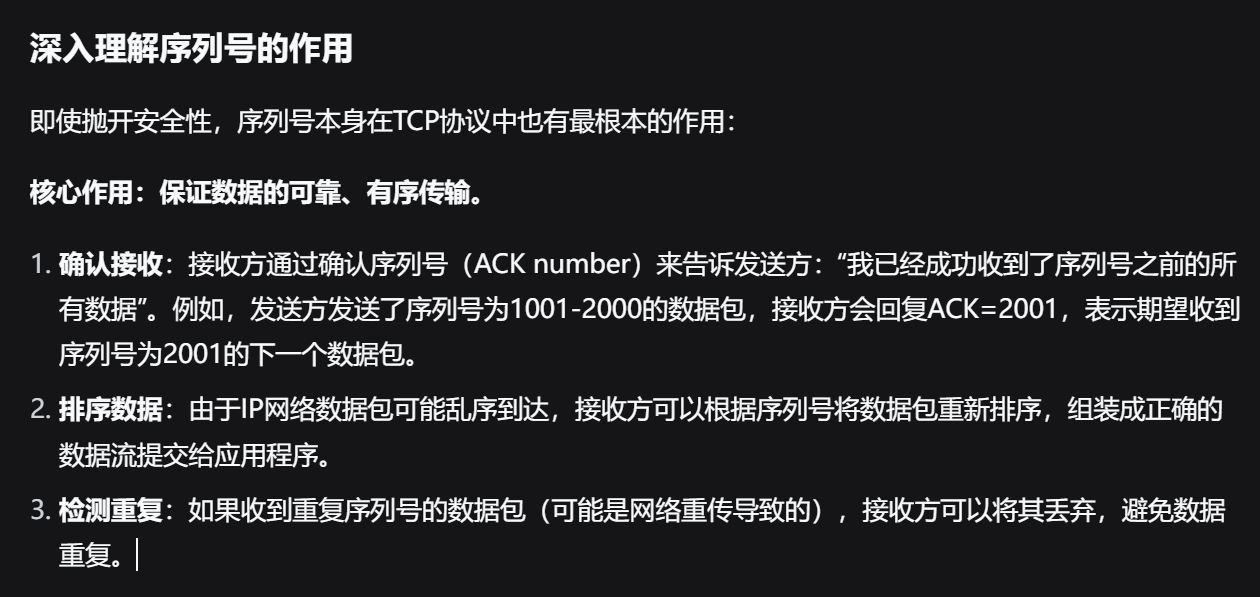
## HTTP怎么实现流量控制（滑动窗口算法）



## TCP每次连接时序列号都一样吗？为什么不一样，有什么作用？







## HTTPS协议和HTTP协议的区别？

（1）数据传输安全性：

http：明文传输，容易被窃听、篡改

https：通过SSL/TSL协议对数据进行加密传输，提供数据机密性和完整性保障。

（2）端口号：

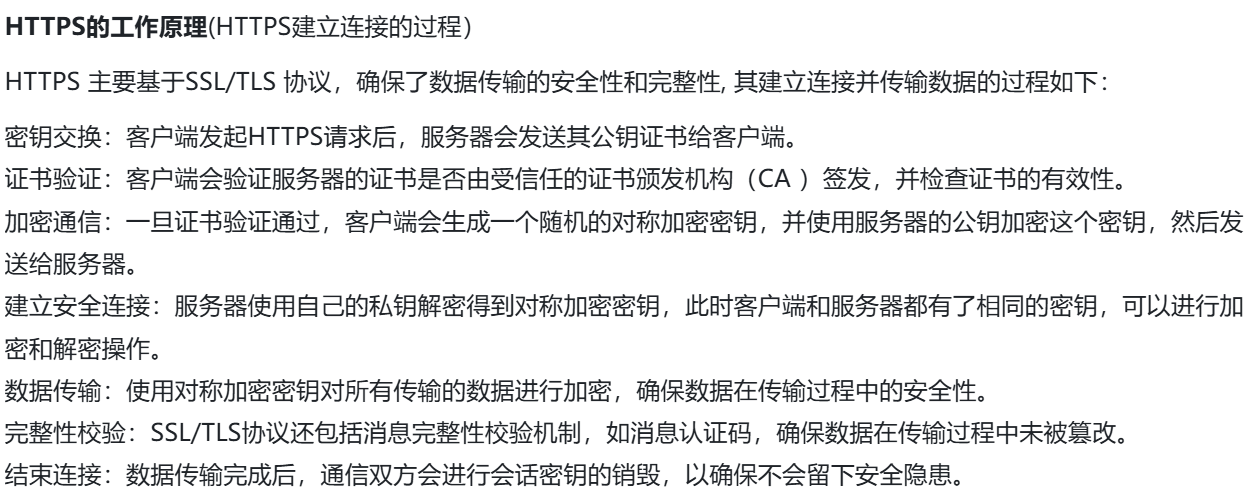
http：默认端口号80

https：默认端口号443

（3）性能：

http：无加密过程，连接建立速度稍快。

https：基于http上又加了SSL或TSL协议来实现的加密传输，加解密过程增加了计算开销，握手时间较长。



## DDOS攻击

DDOS攻击（Distributed Denial of Service，**分布式拒绝服务攻击**）是一种通过大量恶意流量淹没目标服务器、网络或服务，使其无法正常响应合法用户请求的网络攻击方式。

**基本原理**

**拒绝服务（DoS）**：攻击者通过耗尽目标的带宽、计算资源（如CPU、内存）或应用处理能力，导致服务瘫痪。

**分布式（Distributed）**：攻击流量来自全球大量被控制的设备（如僵尸网络中的电脑、IoT设备等），而非单一来源，难以追踪和防御。

**常见攻击类型**

**流量洪泛**  
例如：UDP洪水，通过垃圾流量塞满目标带宽。

**协议攻击**  
例如：SYN洪水（耗尽TCP连接资源）、DNS放大攻击（利用DNS协议缺陷放大流量）。

**应用层攻击**  
例如：HTTP洪水（模拟大量合法请求耗尽服务器资源），更隐蔽且难以识别。

**防御措施**

**流量清洗**：通过云安全服务（如Cloudflare、阿里云高防IP）过滤恶意流量。

**黑名单/IP限速**：识别并拦截异常IP。

**冗余架构**：分布式服务器分散流量压力。

# JVM部分

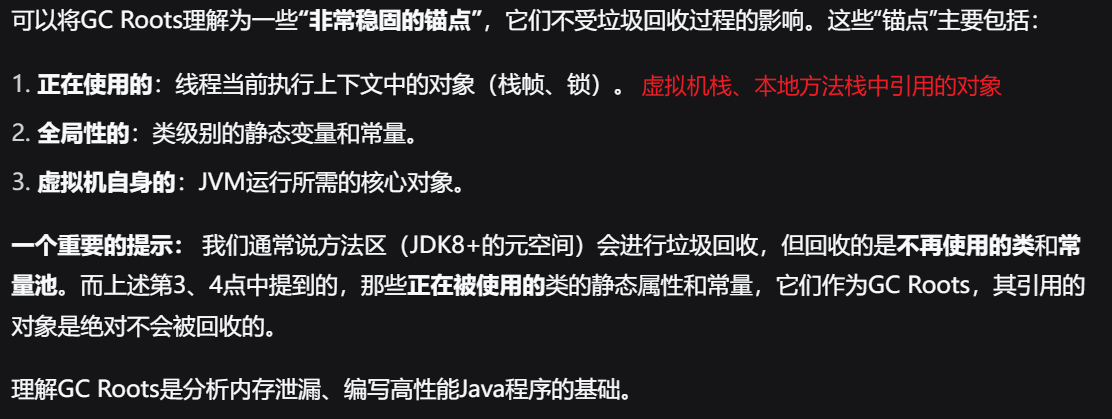
垃圾回收需要知道哪些对象是垃圾（垃圾的搜集算法），主要是两种方式：

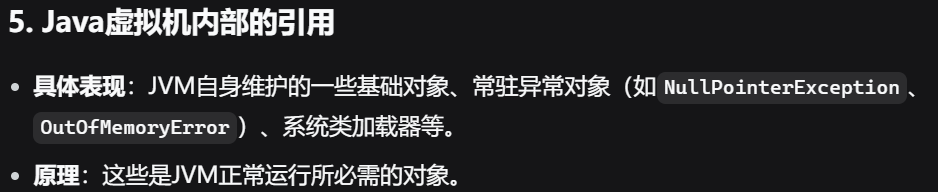
（1）**引用计数法：**每个对象都有一个**引用计数器**，每当有一个引用指向他，计数器就加1，所以GC只要去看对象的计数器是不是0就行，是0的就可以直接回收，但是这个方法最大的问题就是，如果**两个对象相互引用**，那就永远不能被回收，所以就需要方法2

（2）**可达性分析算法**

从**GC Roots**出发，看看哪些对象是可达的，可达的即存在引用，不可达的就可以直接回收（GC Roots可以是栈中引用的对象，类静态属性引用的对象，常量引用的对象以及本地方法引用的对象）

## GC Roots包含哪些对象





标记出垃圾之后就是进行回收了，下面是回收的算法：

## GC垃圾回收算法

**1.标记清除算法**

当垃圾回收器进行内存扫描后会标记出所有的垃圾，然后**直接清除这些带标记**的垃圾对象即可，但是因为垃圾在内存上是分散的，这样清理就会**产生大量的内存碎片**，使得内存利用率越来越低；

**2.复制算法**

**准备两块一模一样的内存空间**，当第一块剩余空间不足时，可以将**所有需要保留的对象拷贝**至另一块空内存，然后将**前一块内存全部清空**，这样即做到了垃圾回收，又做到了碎片整理，但是这样**内存空间就会有一半被浪费**。

**3.标记整理法**

标记整理算法就是在清理垃圾的基础上，多了一步碎片整理的工作，因为整理是比较耗时的，所以显然这种垃圾回收机制不适合高频率的执行。

知道了常见的垃圾回收算法，再介绍下常见的垃圾回收器：

## JVM常见的垃圾回收器

最早期的就是**Serial和SerialOld**，但是这种垃圾回收器是**单线程的**，不支持并发，**开始垃圾回收后所有用户线程都必须暂停**。

后面是**parallel**的垃圾回收器，垃圾回收可以**多线程执行**，但是当开始垃圾回收的时候依然会**触发所有用户线程暂停**。

再之后就是**CMS**，CMS虽然在**初始标记的时候也会触发用户线程暂停**，但是因为CMS**初始标记只会标记第一层的根对象**，所以时间很短，**真正的标记阶段，打标记是和用户线程并行的**；为了避免并行过程中可能的错标，还有第三个**重新标记的阶段**，**这个阶段也会让用户线程暂停，然后通过多线程的方式去检查并且修正错误标记。**但是这个算法也有问题，就是**清理垃圾的时候用户线程也在运行，因此新产生的垃圾没办法及时清理**，需要等下一次回收。

再就是**G1回收器**，G1回收器会**把整个堆内存划分成若干相等大小的区域**，然后**对这些区域进行价值排序**，垃圾越多，回收需要的时间也越多，但是G1回收器可以**根据我们设定的用户线程暂停时间来调整策略**，会尽量**满足我们设定的时间去回收价值高的区域**。另外**G1还把大对象单独存放在了一个区域**，避免了整理时候**频繁移动大对象**。

补充一个新生代和老年代的知识：

## 新生代和老年代？

区分新生代和老年代主要是为了提高垃圾回收效率，大多数的对象存活时间短，很快就会变成垃圾不再使用，这些短生命周期的对象就会分配在新生代；少部分对象长期存活，不会很快被回收，就晋升到老年代。

针对不同的分区的垃圾回收算法也不一样，**新生代通常采用复制算法**，**老年代**通常采用**标记整理算法或标记清除算法**。

## 什么是FullGC？什么情况下触发？怎么解决？

Full GC（完全垃圾回收）是指对整个JVM**堆（包括新生代、老年代）**以及**方法区（元空间）**进行的全面垃圾回收。Full GC会暂停所有应用线程（Stop-The-World），通常耗时更长。

触发的场景：

1. 老年代空间不足，且通过old GC仍然不足
2. 元空间或者永久代内存不足
3. 使用了System.gc()命令
4. 新生代对象要晋升到老年代，但是老年代空间不够

频繁full GC的问题：

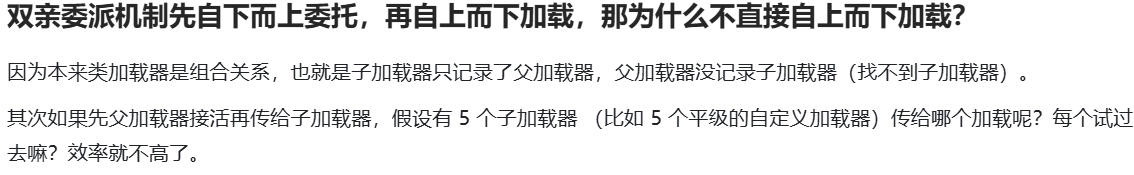
长时间的Stop-The-World暂停会导致应用响应变慢，大量的CPU时间用于GC而非业务处理，也会导致业务吞吐量下降，可能会导致OOM或者服务超时

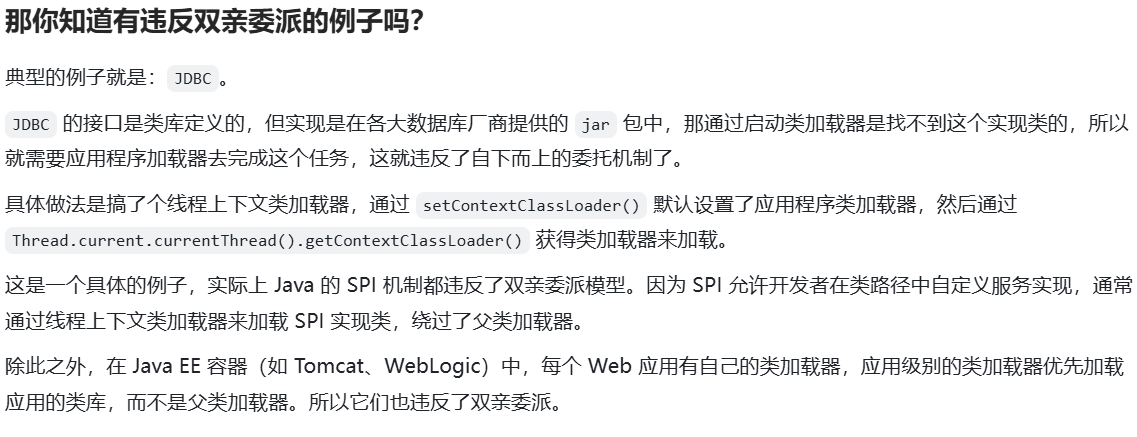
解决：

增加堆大小、调整新生代与老年代的比例、选择合适的垃圾回收器（比如G1替代CMS）、代码优化（及时释放不再使用的对象，减少大对象的频繁创建）

## 双亲委派模型







## JVM的内存区域？

首先是**线程共享**的部分，一共有两个：

一个是**堆（Heap）**，所有**对象实例和数组**都在这里分配内存，垃圾回收器（**GC**）也主要在堆中工作。堆中还包含了**字符串常量池**（String Constant Pool）。

另一个是**元空间（Method Area）**：用于**存储类信息、常量、静态变量、方法字节码**等。其中运行时常量池（Runtime Constant Pool）是元空间的一部分，用于存储编译期生成的各种字面量和符号引用。

### 字符串常量池的作用？

**为什么需要字符串常量池？**

String s1 = "Hello";

String s2 = "Hello";

String s3 = new String("Hello");

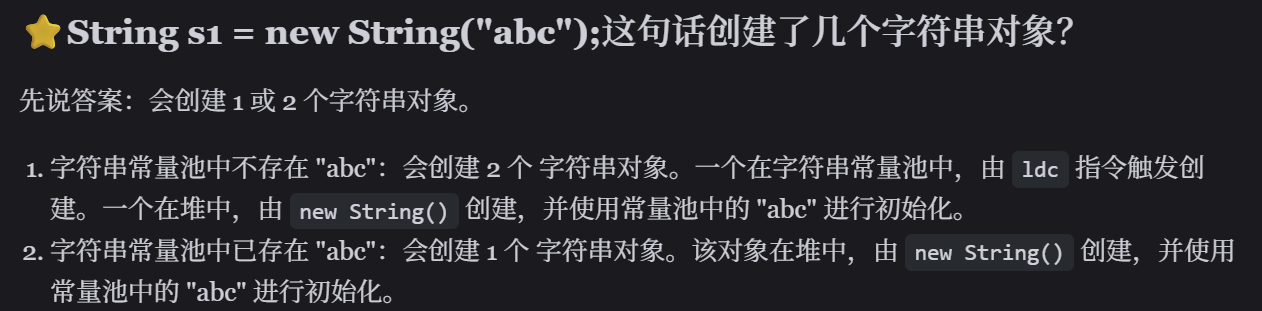
如果没有常量池：s1会创建一个新的String对象。s2会再创建一个内容完全相同的新String对象。s3通过new关键字，毫无疑问也会创建一个新对象。这样，内存中就会有三个内容完全相同的"Hello"对象，这是极大的浪费。

字符串常量池就是为了解决这个问题而生的。**工作原理（核心机制）：**当在代码中直接使用字符串字面量（用双引号包裹）时，例如String s = "Hello";，JVM会首先去字符串常量池中查找是否已经存在一个内容为"Hello"的字符串对象。

**如果存在**：JVM不会创建新的对象，而是直接返回池中已有对象的引用。这样，所有相同的字面量都指向同一个内存地址。

**如果不存在**：JVM会在字符串常量池中创建一个新的String对象，内容为"Hello"，然后返回这个新对象的引用。

new String()**的创建**：当使用new关键字（如String s = new String("Hello");）时，JVM的行为会有所不同：new关键字会**强制**在Java堆的**非常量池区域**创建一个全新的、独立的String对象。这个新对象的内容会初始化为"Hello"，但它和常量池中的那个"Hello"是两个不同的对象。



然后是**线程私有**的部分，一共有三个，

第一个是**虚拟机栈**（VM Stack），**每个线程启动时都会创建一个虚拟机栈**，它存储方法调用过程中产生的栈帧，包括**局部变量、方法返回地址**等，**每个方法调用都会创建一个新的栈帧**，方法执行结束后栈帧出栈。

第二个是**本地方法栈**（Native Method Stack），专门用于存储**本地方法**（Native Method）的调用信息，与虚拟机栈类似，但用于JNI（Java Native Interface）调用。

第三个是**程序计数器**（Program Counter Register），**记录当前线程正在执行的字节码指令地址**。它是JVM运行时最小的内存区域，每个线程都有一个独立的程序计数器。

在JDK1.8时 JVM的内存结构主要有两点不同：

一个是方法区（Method Area）在JDK 1.8被替换为元空间（Metaspace）实现，且元空间使用本地内存（原因：元空间可以动态调整大小，能够避免Out of Memory的错误，并且提高了GC回收效率）

另一个是运行时常量池（Runtime Constant Pool）在JDK1.7属于方法区的一部分，而在JDK 1.8变成元空间的一部分。

## 对象创建的过程了解吗？

第一步是进行**类加载检查**，当程序执行到new指令时，JVM会**先检查对应的类是否已经被加载**、解析和初始化过。如果类尚未加载，JVM会按照类加载机制（加载、验证、准备、解析、初始化）完成类的加载过程。这一步**确保了类的元信息（如字段、方法等）已经准备好**，为后续的对象创建奠定基础。

第二步是进行**内存的分配**，JVM会为新对象分配内存空间。对象所需的内存大小在类加载完成后就可以确定，因此**分配内存的过程就是从堆中划分一块连续的空间**，主要有两种方式：

一种是通过指针碰撞，**如果堆中的内存是规整的**（已使用和空闲区域之间有明确分界），JVM可以**通过移动指针来分配内存**。另一种是通过空闲列表，如果**堆中的内存是碎片化的**，JVM会维护一个**空闲列表**，记录可用的内存块，并从中分配合适的区域。

第三步是将**零值初始化**，JVM会对分配的内存空间进行初始化，将其所有字段设置为零值（如int为0，boolean为false，引用类型为null）。这一步确保了对象的实例字段在未显式赋值前有一个**默认值**，从而避免未初始化的变量被访问。

第四步是**设置对象头**，其中包含Mark Word、Klass Pointer和数组长度。Mark Word用于**存储对象的哈希码、GC分代年龄**等信息。Klass Pointer指向对象所属类的元数据（即Person.class的地址）。

第五步是**执行构造方法**，用**<init>方法完成对象的初始化**。构造方法会根据代码逻辑对对象的字段进行赋值，并调用父类的构造方法完成继承链的初始化。这一步完成后，对象才真正可用。

**一、堆配置：**

**1.1.配置项**

-Xms：初始堆大小

-Xmx：最大堆大小

-XX:NewSize=n：设置年轻代大小

-XX:NewRatio=n：设年轻代和年老代的比值。如：为3表示年轻代和年老代比值为1：3，年轻代占整个年轻代年老代和的1/4，默认为2

-XX:SurvivorRatio=n：年轻代中Eden区与两个survivor区的比值，注意Survivor区有两个，默认8。如：3表示Eden:3 Survivor：2，一个Survivor区占整个年轻代的1/5

-XX:MaxPermSize=n：设置持久代大小

**1.2.说明**

1、一般初始堆和最大堆设置一样，因为：现在内存不是什么稀缺的资源，但是如果不一样，从初始堆到最大堆的过程会有一定的性能开销，所以一般设置为初始堆和最大堆一样。64位系统理论上可以设置为无限大,但是一般设置为4G，因为如果再大，JVM进行垃圾回收出现的暂停时间会比较长，这样全GC过长，影响JVM对外提供服务，所以不能太大。一般设置为4G。

2、-XX:NewRaio和-XX:SurvivorRatio这两个参数，第一是设置年轻代的大小，二个是设置年轻代的比值、理论上设置一个即可以满足需求（因为有默认值）

**1.3. 概念解释**

年轻代：包括Eden区和Survivor区，用于管理新创建的对象。

老年代：用于存放从年轻代中存活下来的对象。

·Eden区：新对象首先被分配到这里。

·Survivor区：用于存放从Eden区中存活下来的对象，通过两个Survivor区的交替使用减少内存碎片。

**年轻代(Young Generation)**

年轻代是对象最初被分配的地方。大多数对象在年轻代中创建，并且大多数对象在年轻代中死亡(即不再被引用)。年轻代的主要特点是：

Eden区：新创建的对象首先被分配到Eden区。Eden区是年轻代的主要部分，通常占据年轻代的大部分空间。

Survivor区：Survivor区分为两个部分，通常称为From Survivor区和To Survivor区。在每次Minor GC(年轻代垃圾回收)后，存活的对象会被移动到Survivor区中的一个，而另一个Survivor区则被清空。这种设计有助于减少内存碎片。(复制算法)

**老年代(old Generation)**

老年代用于存放从年轻代中存活下来的对象。当对象在年轻代中经过多次Minor GC后仍然存活，它们会被晋升到老年代。老年代的特点是：

Major GC(Full GC)：老年代的垃圾回收称为Major GC或Full GC。Full GC通常比Minor GC更耗时，因为它需要扫描整个堆内存。

**Eden区与Survivor区**

Eden区：新对象首先被分配到Eden区。当Eden区满时，会触发一次Minor GC，将存活的对象移动到Survivor区中的一个(通常是From Survivor区)。

Survivor区：Survivor区用于存放从Eden区中存活下来的对象。在每次Minor GC后，存活的对象会被移动到另一个Survivor区(To Survivor区)，而原来的Survivor区(From Survivor区)则被清空。这种设计有助于减少内存碎片，提高内存利用率。

**二、调优总结**

**年轻代大小选择：**

**响应时间优先的应用：**尽可能设置大，直到接近系统的最低响应时间限制(根据实际情况选择)。在此种情况下，年轻代收集发生的频率也是最小的。同时减少到达年老代的对象。

**吞吐量优先的应用：**尽可能的设置大，可能到达Gbit的程度，因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8核CPU以上应用。

**年老代大小选择：**

**响应时间优先的应用：**年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可能会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

1、并发垃圾收集信息

2、持久代并发收集次数

3、传统GC信息

4、花在年轻代和年老代回收上的时间比例，减少年轻代和年老代花费的时间，一般会提高应用的效率。

**吞吐量优先的应用：**一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期对象，而年老代尽存放长期存活的对象

**较小堆引起的碎片问题：**

因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

-XX：+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩

-XX：CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次FullGc后，对年老代进行压缩。

**三、内存泄露检查**

根据垃圾回收前后情况对比，同时根据对象引用情况（常见的集合对象引用）分析，基本都可以找到泄漏点。

**持久代占满处理：**

1、-XX：MaxPermSize=16m设置持久代大小

2、换JDK、比如：JRocket

**系统内存被占满：**

一般是因为没有足够的资源产生线程造成的，系统创建线程时，除了要在Java堆中分配内存外，操作系统本身也需要分配资源来创建线程。因此，当线程数量大的一定程度以后，堆中或许还有空间，但是操作系统分配不出资源来了，出现异常；

分配给Java虚拟机的内存越多，系统剩余的资源就越少，因此，当系统内存固定时，分配给Java虚拟机的内存越多，那么，系统总共能够产生的线程也就越少，两者成反比。同时，可以通过修改-Xss来减少分配给单个线程的空间，也可以增加系统总共生产的线程数。

**四、GC使用与性能优化管理**

(1)不要显式调用System.gc()。此函数建议JVM进行主动GC，虽然只是建议而非一定，但很多情况下它会触发主GC，从而增加主动GC的频率、也即增加了间歇性停顿的次数。大大的影响系统性能。

(2)尽量减少临时对象的使用。临时对象在跳出函数调用后，会成为垃圾，少用临时变量就相当于减少了垃圾的产生，从而减少了主GC的机会。

(3)对象不用时最好显式置为Null。一般而言，为Null的对象都会被作为垃圾处理，所以将不用的对象显式地设为Null，有利于GC收集器判定垃圾，从而提高了GC的效率。

(4)尽量使用StringBuffer，而不用String来累加字符串。由于String是固定长的字符串对象，累加String对象时，并非在一个String对象中扩增，而是重新创建新的String对象，如Str5=Str1+Str2+Str3+Str4，这条语句执行过程中会产生多个垃圾对象，因为对次作“+”操作时都必须创建新的String对象，但这些过渡对象对系统来说是没有实际意义的，只会增加更多的垃圾。避免这种情况可以改用StringBuffer来累加字符串，因StringBuffer是可变长的、它在原有基础上进行扩增，不会产生中间对象。

(5)能用基本类型如int，long，就不用Integer，Long对象。基本类型变量占用的内存资源比相应对象占用的少得多，如果没有必要，最好使用基本变量。

(6)尽量少用静态对象变量。静态变量属于全局变量，不会被GC回收，它们会一直占用内存

(7)注意分散对象创建或删除的时间，集中在短时间内大量创建新对象，特别是大对象，会导致突然需要大量内存、JVM在面临这种情况时，只能进行主GC，以回收内存或整合内存碎片，从而增加主GC的频率。集中删除对象，道理也是一样的。

**五、JVM调优参数参考**

1.针对JVM堆的设置，一般可以通过-Xms-Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，通常把最大、最小设置为相同的值;

2.年轻代和年老代将根据默认的比例(1:2)分配堆内存，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代 比如年轻代，通过

-XX：newSize-XX：MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX：newSize-XX：MaxNewSize设置为同样大小。

3.年轻代和年老代设置多大才算合理

1)更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间;小的年老代会导致更频繁的FullGC

2)更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短：大的年老代会减少FullGC的频率

如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代;如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性。

在抉择时应该根据以下两点：

(1)本着FullGC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理。

(2)通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响FullGC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间。

4.在配置较好的机器上（比如多核、大内存），可以为年老代选择并行收集算法XX+UseParallelOldGC。

5.线程堆栈的设置：每个线程默认会开启1M的堆栈，用于存放栈帧、调用参数、局部变量等，对大多数应用而言这个默认值太了，一般256K就足用。

理论上，在内存不变的情况下，减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

**六、触发Full GC的几种情况**

**1.老年代空间不足**

（1）年轻代晋升：当年轻代中的对象经过多次Minor GC后仍然存活，会被晋升到老年代。如果老年代空间不足，无法容纳这些晋升的对象，就会触发Full GC。

（2）大对象直接分配：如果应用程序直接分配大对象(超过年轻代Eden区的大小)，这些对象会直接进入老年代。如果老年代空间不足，也会触发Full GC。

**2.方法区(元空间或永久代)空间不足**

（1）类加载：如果应用程序动态加载大量类，导致方法区(Metaspace或永久代)空间不足，会触发Full GC。

（2）元数据回收：Metaspace或永久代中的元数据(如类信息、方法信息等)不再被引用时，需要通过Full GC进行回收。

**3.System.gc()调用**

显式调用：应用程序中显式调用System.gc()或Runtime.getRuntime().gc()会建议JVM执行Full GC。不过JVM不一定会立即执行，具体行为取决于JVM的实现和配置。

**4.垃圾回收器策略**

（1）CMS GC的并发模式失败：在使用CMS(Concurrent Mark-Sweep)垃圾回收器时,如果在并发标记过程中老年代空间不足，会触发Full GC。

（2）G1 GC的疏散失败：在使用G1(Garbage-First)垃圾回收器时，如果在疏散(Evacuation)阶段无法找到足够的空闲区域来存放存活对象，会触发Full GC。

**5.堆内存分配失败**

内存泄漏：如果应用程序存在内存泄漏，导致堆内存中大量对象无法被回收，最终堆内存耗尽，会触发Full GC

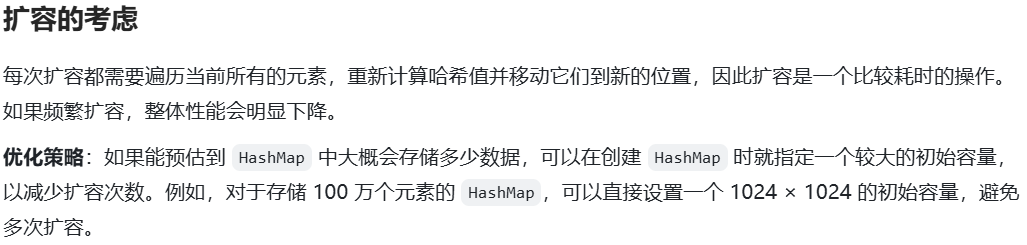
# Java集合部分

## HashMap的原理

HashMap在jdk1.7和1.8实现上是有些不一样的，先介绍1.7。在1.7中，底层是通过**数组+链表**来实现的，当我们插入元素的时候，会计算key的hash值，也就是**对数组长度取模**，得到插入位置的下标，如果该位置为空，就插入元素，**如果不为空，就会以链表的形式存储**，会去遍历链表，如果找到了相同的key，就替换value，表示修改操作；如果没有相同的key，就把新的entry**插入到链表的头部**。在1.7中，扩容机制是，比如默认数组容量是16，加载因子0.75，所以当我们插入第13个元素的时候就会触发扩容，**扩容会重新对所有元素进行hash计算，去把元素放到新的数组中去**。

但是1.7中存在一些问题：首先就是**哈希冲突比较严重的时候链表会变得很长**，链表的查询效率是O(n)，这就会影响性能。另外，**头插法虽然比较快，但是在多线程环境可能就会形成环形链表，陷入死循环**（比如现在链表是A->B->C，线程A和线程B同时去操作链表，如果线程A先去操作时候发现需要扩容，通过头插法扩容A先放入新数组，然后是B和C，顺序就变成了C->B->A，这个时候线程B再去操作，线程B还以为是原链表，即A指向B，但是现在实际已经变成了B指向A，就形成了死循环）；再就是扩容，1.7是对所有元素重新计算，这个也比较复杂。

针对这3个问题，首先一点，1**.8改成了尾插法**，**扩容不会反转链表**，所以避免了死循环的产生；第二点，**1.8中采用数组+链表+红黑树的结构**。当我们插入数据，如果对应位置已经有元素，会先存储成链表，但是如果**链表的长度已经等于8**了，就需要看**数组长度是否大于64**，**如果没有就优先扩容数组**；**如果数组元素已经大于64，就把链表转换成红黑树存储**，这个转换的契机就是链表长度大于8，然后**如果元素少于6个，就从二叉树再退化回链表**。1.8里的扩容契机和1.7一样，除了前面提到的链表长度那里以外，也是判断数组存储的元素是否超过临界值。但是1.8的扩容不是全部重新计算hash，而是**通过元素的hash和老数组的长度进行&运算来计算出元素是处于高位还是低位**，**如果结果是0就把元素留在原来位置不移动，否则就移动到原来索引位置加上老数组容量的位置去**，显著提升了扩容的速度。



注意：声明初始容量的时候需要考虑集合的实现类型，如果是hashmap或者hashset（实际就是hashmap，只不过value为空），那初始容量实际要声明成100W/扩容因子（默认0.75），如果是非hashmap实现，比如arraylist，直接声明成100W就可以了，因为arraylist是存满再扩容，没有扩容因子。

## ArrayList和LinkedList的区别？



## 常见的集合有哪些？



## 1亿量级的ArrayList数据去重？

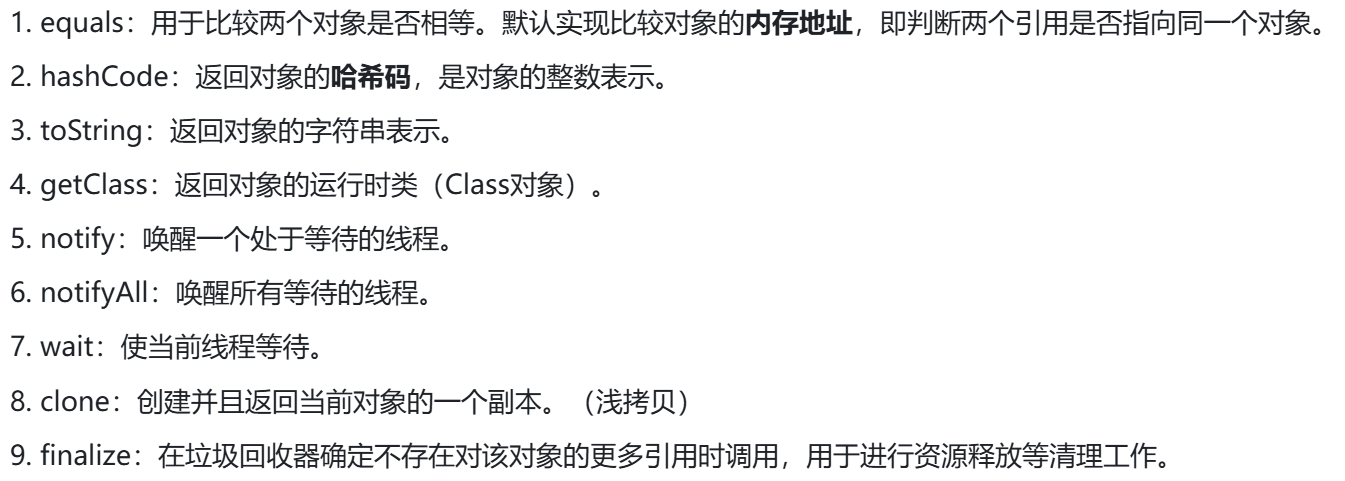
（1）直接hashset，但是注意初始化容量，避免频繁扩容（可以初始化成ArrayList.size() / 0.75 + 1）

（2）排序+遍历去重，排序完之后，一遍扫过去，只要和前一个不一样就保留；

（3）bitmap，先遍历一遍arraylist，用一个二进制位去把arraylist里元素值对应位置的bit值改为1，遍历完再遍历位图，收集所有标记为1的位置下标（如果是0-1亿之间的数字，bitmap只需要12MB的内存，如果是int范围512MB也够了）

# Java基础部分

## Java Object类中有什么方法？



## 深拷贝和浅拷贝？

深拷贝和浅拷贝的核心区别在于是否递归地复制对象内部的引用类型数据。

首先是定义上的区别，

浅拷贝是指创建一个新对象，但**新对象中的引用类型字段仍然指向原对象中引用类型的内存地址**。换句话说，浅拷贝只复制了对象本身，而没有复制对象内部的引用类型数据。**修改新对象中的引用类型数据会影响原对象**。

深拷贝是指创建一个新对象，并且**递归地复制对象内部的所有引用类型数据**。换句话说，深拷贝不仅复制了对象本身，还复制了对象内部的所有引用类型数据。**修改新对象中的引用类型数据不会影响原对象**。

其次是实现方式上的区别，

浅拷贝可以使用Object类的**clone()方法**，也可以使用实现Cloneable接口并重写clone()的方法。

深拷贝可以手动对引用类型字段进行**递归拷贝**，也可以使用**序列化**（Serialization）的方式将对象序列化为字节流，再**反序列化**为新对象。

最后是使用场景上的区别，

浅拷贝适用于当对象内部的引用类型数据不需要独立复制的情况。

深拷贝适用于当对象内部的引用类型数据需要完全独立的情况。

## 反射机制的使用场景？

反射的主要作用就是允许程序在运行时动态地获取类的信息并操作类或对象。

反射常用的场景包括：

**1.框架开发 (Frameworks)**

这是反射最经典和最重要的应用场景。框架通常要处理未知的、由用户定义的类，它无法在编译时知道这些类的具体信息。反射使得框架能够动态地加载、检查和调用用户代码，从而实现高度的灵活性和可扩展性。

* **解释**：框架通过配置文件（如XML）、注解（Annotation）或约定来获取用户定义的类名和方法名。在运行时，框架使用反射API（如Class.forName("ClassName")）来加载这个类，创建其实例（clazz.newInstance()），并调用其特定方法（method.invoke(object)）。
* **典型例子**：

Spring: 依赖注入（IoC）容器通过读取配置或扫描注解，使用反射来创建和管理Bean（对象），并自动将依赖关系注入到对应的属性或构造函数中。

MyBatis: ORM框架使用反射将数据库查询结果集（ResultSet）动态地映射到Java实体对象（POJO）的属性上，反之亦然。

JUnit: 测试框架通过反射查找带有@Test注解的方法，并执行它们。

**2.动态代理 (Dynamic Proxy)**

动态代理是一种强大的设计模式，允许在运行时动态地创建一个实现了一组接口的代理类。这个代理类可以将所有的方法调用转发到一个统一的“调用处理器”中处理。

* **解释**：代理的逻辑是通用的，不依赖于某个具体的接口。反射使得动态生成代理类成为可能。当代理对象上的方法被调用时，JVM会将其路由到InvocationHandler.invoke()方法，该方法可以通过反射获取被调用方法的信息（方法名、参数等），并执行额外的逻辑（如日志记录、事务管理、权限检查等）。
* **典型例子**：

Spring AOP (面向切面编程): 其底层就是使用动态代理（JDK动态代理或CGLib）来实现的。它为Bean创建代理，从而能够在目标方法执行前后插入诸如事务、日志、安全等“切面”逻辑。

**3.注解处理 (Annotation Processing)**

虽然编译时注解处理（APT）很常见，但运行时注解处理同样依赖反射。

* **解释**：程序在运行时可以通过反射API（如**getAnnotation()**, **getDeclaredAnnotations()**）来检查类、方法、字段上的注解，并根据注解的元数据执行相应的操作。
* **典型例子**：

Web服务: 使用@Path, @GET, @POST等注解来定义RESTful接口。框架在启动时会扫描这些注解，并通过反射来将HTTP请求映射到对应的Java方法上。

序列化/反序列化: 库如Jackson/Gson在将JSON转换为对象时，会使用反射检查字段上的@JsonProperty等注解，以确定JSON字段和Java字段的对应关系。

**4. 开发工具和集成开发环境(IDEs)**

你的IDE能提供代码自动完成、显示类的方法列表、进行类型检查等功能，这些都离不开反射。

* **解释**：IDE在后台利用反射来分析项目所依赖的库（JAR包），获取其中所有类的结构信息，从而为你提供智能提示和代码洞察功能。

**5. 数据库驱动程序**

JDBC（Java数据库连接）是使用反射的另一个经典案例。

* **解释**：在早期使用Class.forName("com.mysql.cj.jdbc.Driver")来注册数据库驱动时，就是通过反射来动态加载驱动类。驱动类在初始化时会向DriverManager注册自己，使得Java程序能够与特定品牌的数据库进行通信。

**6. 适配和扩展机制**

当程序需要动态加载和使用第三方插件或模块时，反射是必不可少的技术。

* **解释**：主程序可以定义一个统一的接口。第三方开发者按照这个接口实现他们的功能并打包成JAR。主程序通过反射在运行时加载这些JAR包，创建接口实现的实例，并调用其功能，从而实现功能的可插拔式扩展。

（直接调用JAR包是静态调用，必须事先清楚实现的类是什么，而这里用反射，我们并不需要知道实现的类是什么，只需要知道他存放的位置，并且他实现了统一的接口，我就可以读取你JAR包里的配置文件拿到全类名，从而动态的加载）

# Linux常见指令

**文件与目录操作：**

目录切换：cd

目录操作：ls显示目录中的文件和子目录的列表

mkdir：创建新目录

rm：删除文件或目录

cp：复制文件或目录，cp -r用于递归复制目录

find 支持按照文件名、大小、修改时间等条件查找文件或目录

find使用举例：find . -name "\*.txt" 查找当前目录下的所有.txt文件

**文件内容查看：**

cat：查看文件内容（常用于查看小型文本文件）

more/less：分页查看文件内容，less支持向上翻页，适合查看大文件

tail：查看文件末尾的若干行

head：查看文件的开头几行

**系统管理：**

top：查看系统的CPU使用率、内存使用率或者进程

ps：查看当前运行的进程列表 ps -aux查看所有用户的进程

kill：中止进程

**网络配置与调试：**

ping：测试与目标主机的网络连接

ip：查看和配置网络接口信息。

grep命令：用于在文件中搜索特定的字符串或正则表达式

比如：查找文件log.txt中包含error的行

grep "error" log.txt

**进程管理与系统性能优化：**

nice/renice命令：用于调整进程的优先级，nice用于启动进程时设置优先级，renice用于调整运行中的进程优先级，较小的nice值代表更高的优先级。

**文件压缩与解压缩：**

tar：打包/解压压缩包

例如：打包目录mydir并压缩成tar.gz文件：

tar -czvf mydir.tar.gz mydir

解压时使用-x参数：

tar -xzvf mydir.tar.gz

zip/unzip：zip用于压缩文件或目录，unzip用于解压缩

zip -r myfile.zip mydir

unzip myfile.zip

## 根据进程名搜索进程ID？

使用pgrep命令来根据进程名称查找进程的PID

pgrep <进程名称>

## 如何查看服务器的负载情况，使用什么命令？

可以使用top命令

## 什么是物理地址，什么是逻辑地址？

物理地址：计算机内存中真正的内存单元地址，由内存管理单元直接访问，表示数据在物理内存中的实际存储位置，是由硬件层面决定的；物理地址直接对应到内存芯片上的某个位置，它是CPU在访问内存时经过地址转换后的实际地址。

逻辑地址（虚拟地址）：逻辑地址是程序在运行时看到的地址空间，由CPU生成，用于内存的访问和操作，逻辑地址在程序编写和编译时使用，并由操作系统通过地址转换机制映射到物理地址。每个进程拥有独立的逻辑地址空间，从而实现了进程间的内存隔离。

## 虚拟内存？

虚拟内存主要是用来扩展实际可用的物理内存并隔离进程的地址空间，保证安全。比如我们的物理内存是8G，但是我们可以通过虚拟内存支持16GB的进程需求。主要是通过把磁盘空间作为扩展，当物理内存不足，就会把不活跃的页换出到磁盘，等需要的时候再换入。每个程序都运行在独立的虚拟地址空间，就可以防止恶意程序访问其他进程或者系统内存，能够提高安全性。

# SSM框架部分

## Spring IOC

IoC即控制反转。

例如：现有类A依赖于类B

传统的开发方式：往往是在类A中手动通过new关键字来new一个B的对象出来；使用IoC思想的开发方式：不通过new关键字来创建对象，而是通过IoC容器(Spring框架)来帮助我们实例化对象。我们需要哪个对象，直接从IoC容器里面去取即可。

从以上两种开发方式的对比来看：我们“丧失了一个权力” (创建、管理对象的权力)，从而也得到了一个好处（不用再考虑对象的创建、管理等一系列的事情）

为什么叫控制反转?

控制：指的是对象创建（实例化、管理）的权力

反转：控制权交给外部环境（IoC容器）

### IoC解决了什么问题？

IoC的思想就是对象之间不互相依赖，由第三方容器来管理相关资源。这样有什么好处呢？

1、降低了对象之间的耦合度或者说依赖程度；

2、资源变的容易管理；比如用Spring容器提供的话很容易就可以实现一个单例。

例如：现有一个针对User的操作，利用Service和Dao两层结构进行开发，在没有使用IoC思想的情况下，Service层想要使用Dao层的具体实现的话，需要通过new关键字在UserService的实现类中手动new出UserDao的具体实现类。如果后续接到新的需求，针对UserDao接口需要开发另一个新的实现类，我们就需要手动修改UserService实现类中new的对象。如果有许许多多的地方都引用了UserDao的具体实现的话，那修改起来就会非常的繁琐。



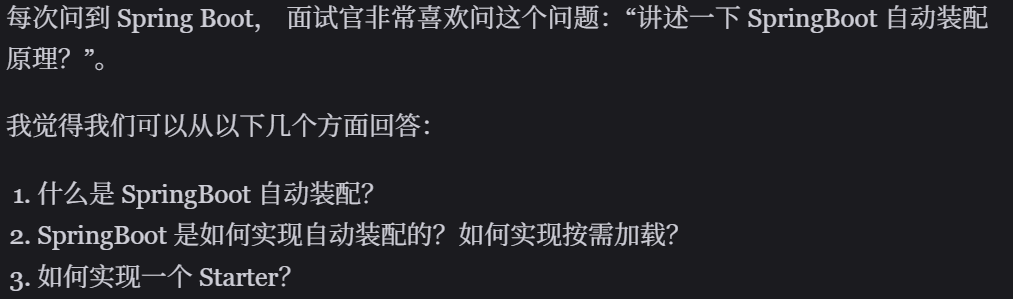
使用IoC的思想，我们将对象的控制权（创建、管理）交由IoC容器去管理，我们在使用的时候直接向IoC容器 “要”就可以了（通过注解去标识，按照类型/名称去进行注入）。

### IoC和DI有区别么？

IoC是一种设计思想或者说是某种模式。这个设计思想就是**将原本在程序中手动创建对象的控制权交给第三方比如IoC容器。**对于我们常用的Spring框架来说，IoC容器实际上就是个Map（key，value），Map中存放的是各种对象。

IoC最常见的实现方式叫做依赖注入简称DI。

## Spring自动装配原理



通过注解或者一些简单的配置就能在Spring Boot的帮助下快速实现某块功能。

在传统Spring中，我们需要在XML或Java配置中显式地定义很多Bean（如数据源等）。而在Spring Boot中，只引入了特定的依赖，Spring Boot就会自动配置好这些组件。举个例子：要使用JDBC，只需在pom.xml中引入spring-boot-starter-jdbc依赖，并在yml文件中配置一些连接信息即可。

**自动装配的实现：**

核心机制：条件化装配

这是自动装配的基石。Spring Boot不会盲目地配置所有东西，它只在某些**条件满足**的情况下才进行配置。这是通过一系列@Conditional注解来实现的。

条件注解：

1. 如果没有引入web启动依赖，所有与web相关的自动配置类都不会被加载；
2. 用户自定义的加了@Bean注解的都会被优先加载；
3. 在yml等配置文件中我们可以自定义配置属性，比如内置的tomcat启动端口是8080，如果我们配置成其他端口，就会按照我们的配置进行自动装配。

自动装配的过程可以概括为以下几个步骤：

1. **启动注解：**@SpringBootApplication

每个Spring Boot主类上都标有@SpringBootApplication，它是一个复合注解，其中@EnableAutoConfiguration注解就启用自动配置的。

1. **启用自动配置：**@EnableAutoConfiguration

这个注解的作用是启用Spring Boot的自动配置机制。它背后是通过一个import选择器的类实现的(AutoConfigurationImportSelector.class)。

1. **加载自动配置列表：**

会通过import选择器类读Classpath下所有JAR包中的META-INF/spring.factories文件。

1. **关键文件：**META-INF/spring.factories

这个文件是一个标准的Java配置文件，内容是**键=值**对的形式。这个列表定义了**所有可能被自动配置的类**。

1. **过滤与条件判断：**

Import选择器不会直接加载所有可能的配置类。而是通过**条件注解**对列表中的配置类进行筛选。只有满足条件的配置类，才会被真正解析，将其中的Bean定义加载到Spring容器中。

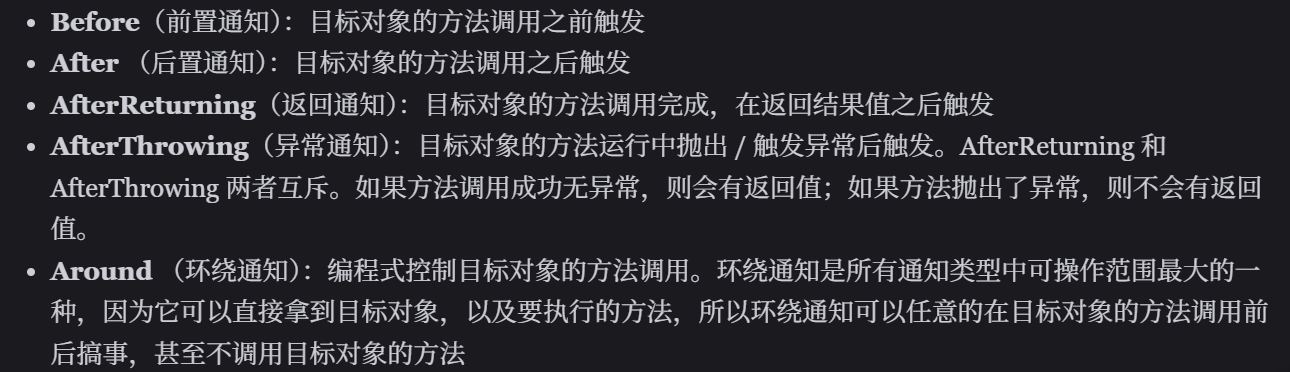
## Spring AOP

AOP（Aspect Oriented Programming）即面向切面编程，AOP是OOP（面向对象编程）的一种延续，二者互补，并不对立。

AOP的目的是将一些分散在多个类或对象中的公共行为（如日志记录、事务管理、权限控制、接口限流等）从核心业务逻辑中分离出来，通过动态代理技术，实现代码的复用和解耦，提高代码的可维护性和可扩展性。

AOP之所以叫面向切面编程，是因为它的核心思想就是将横切关注点从核心业务逻辑中分离出来，形成一个个的切面（Aspect）。

### AOP常见的通知（增强）类型



### AOP解决了什么问题？

OOP不能很好地处理一些分散在多个类或对象中的公共行为（如日志记录、事务管理、权限控制、接口限流、接口幂等等），这些行为通常被称为横切关注点。如果我们在每个类或对象中都重复实现这些行为，那么会导致代码的冗余、复杂和难以维护。

AOP可以将横切关注点（如日志记录、事务管理、权限控制、接口限流、接口幂等等）从核心业务逻辑中分离出来，实现关注点的分离。

比如日志记录，没有AOP之前，我们需要对需要加日志记录的地方挨个去写代码，全是重复的逻辑；但是有AOP技术之后，我们就可以把日志记录的逻辑封装成一个切面，然后通过切点和通知来指定具体在哪些方法中加入日志。在指定方法中只需要加一行注解就可以实现日志记录。

### AOP的应用场景

**日志记录：**自定义日志记录注解，利用AOP，给业务方法上加一行注解即可实现日志记录。

**性能统计：**利用AOP在目标方法的执行前后统计方法的执行时间，方便优化和分析。

**事务管理：**@Transactional注解可以让Spring为我们进行事务管理比如回滚异常操作，免去了重复的事务管理逻辑。@Transactional注解就是基于AOP实现的。

**权限控制：**利用AOP在目标方法执行前判断用户是否具备所需要的权限，如果具备，就执行目标方法，否则就不执行。

**接口限流：**利用AOP在目标方法执行前通过具体的限流算法对请求进行限流处理。

### AOP动态代理

Spring AOP是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用**JDK代理**，去创建代理对象，而对于没有实现接口的对象，就无法使用JDK去进行代理了，这时候Spring AOP会使用CGLIB生成一个被代理对象的子类来作为代理。

Spring Boot默认使用CGLIB动态代理。

## Spring事务

事务是逻辑上的一组操作，要么都执行，要么都不执行。

我们系统的每个业务方法可能包括了多个原子性的数据库操作，比如转账就是经典的事务场景。事务能否生效数据库引擎是否支持事务是关键。比如常用的MySQL数据库默认使用支持事务的innodb引擎。但是，如果把数据库引擎变为myisam，那么程序也就不再支持事务了！

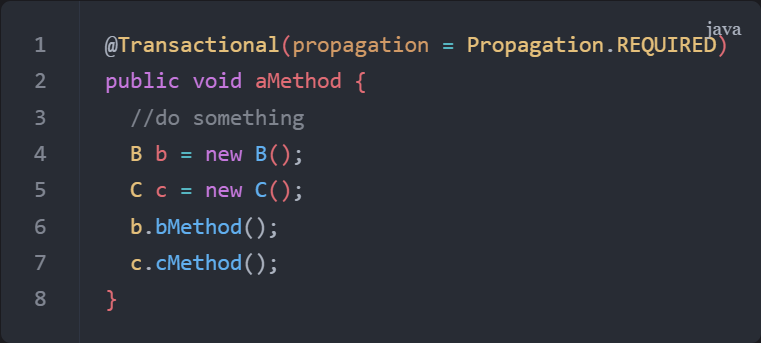
### Spring对事务的支持

（1）编程式事务管理

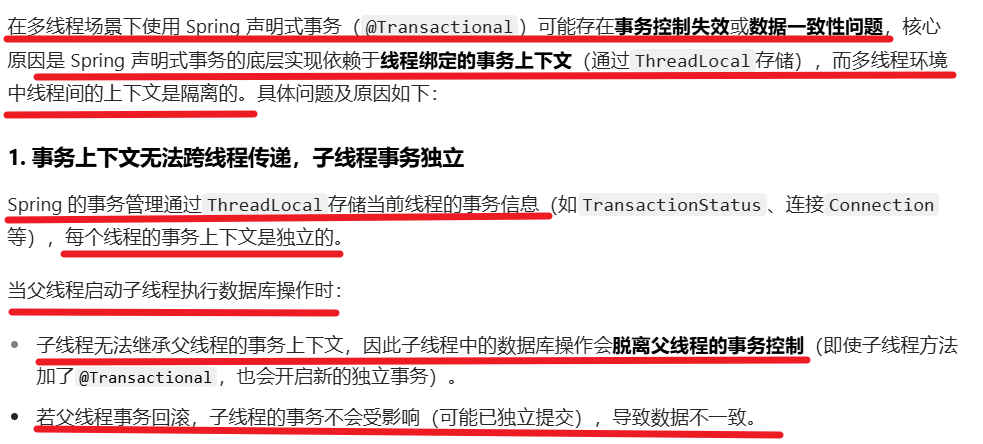
通过TransactionTemplate或者TransactionManager手动管理事务，实际应用中很少使用，但是对于你理解Spring事务管理原理有帮助。

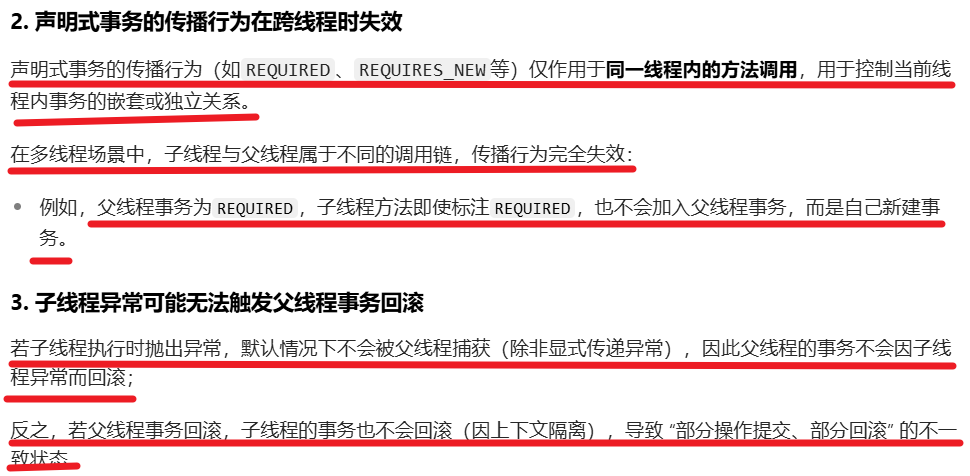
（2）声明式事务管理

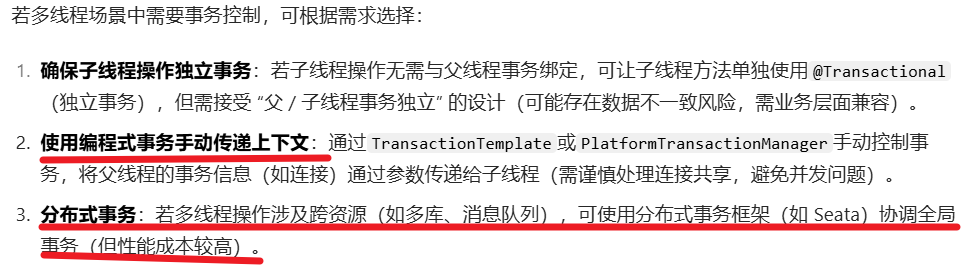
推荐使用（代码侵入性最小），实际是通过AOP实现（基于@Transactional的全注解方式使用最多）。



### 声明式事务在多线程场景下的问题







### 事务的属性

事务属性包含了5个方面：隔离级别、传播行为、回滚规则、是否只读、事务超时

**只读模式**

@Transactional(readOnly = false)

只读模式可以提升查询事务的效率，推荐事务中只有查询代码时，使用只读模式。默认是false，一般情况下，都是在类上添加@Transactional注解，针对类下的查询方法可以通过再次添加@Transactional注解，设置为只读模式，从而提高查询的效率（因为查询并不会改变数据库的数据，所以本身就不需要事务）。

**超时时间**

@Transactional(timeout = 3)

默认值是-1，代表永远不超时，设置timeout = 时间（秒数），超过时间，就会回滚事务和报异常（TransactionTimedOutException），如果类上设置了，方法也设置了事务注解，方法上的注解会覆盖掉类上的注解！

**指定事务异常回滚规则**

@Transactional(rollbackFor=Exception.class, noRollbackFor=FileNotFoundException.class)

默认只针对运行时异常回滚（error也会回滚），编译时异常不回滚。可以指定：

rollbackFor属性：指定哪些异常类才会回滚，默认是RuntimeException and Error 异常方可回滚；

noRollbackFor属性：指定哪些异常不会回滚，默认没有指定，如果指定，应该在rollbackFor的范围内!

为了让发生所有异常都进行事务的回滚，我们可以指定Exception异常来控制所有异常都回滚！即：rollbackFor = Exception.class

**事务隔离级别**

*@Transactional(isolation = Isolation.READ\_COMMITTED)*

Spring支持的隔离级别枚举：

Isolation.DEFAULT：使用数据库默认隔离级别。

Isolation.READ\_UNCOMMITTED：读未提交。

Isolation.READ\_COMMITTED：读已提交。

Isolation.REPEATABLE\_READ：可重复读。

Isolation.SERIALIZABLE：串行化。

数据库事务的隔离级别是指在多个事务并发执行时，数据库系统为了保证数据一致性所遵循的规定。常见的隔离级别包括：

* 读未提交（Read Uncommitted）：事务可以读取未被提交的数据，容易产生脏读、不可重复读和幻读等问题。实现简单但不太安全，一般不用。
* 读已提交（Read Committed）：事务只能读取已经提交的数据，可以避免脏读问题，但可能引发不可重复读和幻读。（大多数数据库的默认隔离级别）
* 可重复读（Repeatable Read）：确保在同一事务中多次读取同一数据时，结果一致，不管其他事务对数据做了什么修改。可以避免脏读和不可重复读，但仍有幻读的问题。（MySQL的默认隔离级别）
* 串行化（Serializable）：最高的隔离级别，完全禁止了并发，只允许一个事务执行完毕之后才能执行另一个事务。可以避免以上所有问题，但效率较低，不适用于高并发场景。

**事务传播行为**

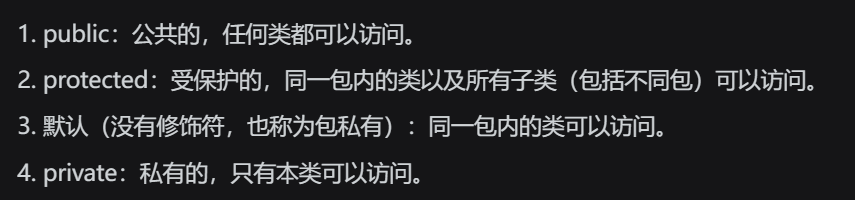
*@Transactional(propagation = Propagation.REQUIRED)*

事务传播行为定义了**多个事务方法相互调用时，事务应该如何传播**。Spring提供了7种事务传播行为，通过@Transactional注解的propagation属性进行配置。

| **传播行为类型** | **说明** |
| --- | --- |
| REQUIRED（默认） | 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新事务，保证最后仅有一个事务。 |
| REQUIRES\_NEW | 无论当前是否存在事务，都创建一个新事务，并挂起当前事务（挂起事务的意思是在当前事务执行过程中，暂时将其暂停，并开启一个新的事务；挂起事务后，当前事务的状态会被保存，直到新事务执行完毕后再恢复），最后会有多个独立的事务，所以如果后面的事务报异常了，前面事务已经修改的数据并不会跟着一起回滚，适用于独立事务的场景，比如日志记录。 |
| SUPPORTS | 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务方式执行，适用于查询方法，不需要强制事务。 |
| NOT\_SUPPORTED | 以非事务方式执行操作，如果当前存在事务，则挂起该事务，适用于不需要事务支持的操作。 |
| MANDATORY | 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常，强制要求必须开启事务。 |
| NEVER | 以非事务方式执行操作，如果当前存在事务，则抛出异常，强制要求不能开始事务。 |
| NESTED | 如果当前存在事务，则在嵌套事务内执行；如果当前没有事务，则创建一个新事务，适用于需要部分回滚的场景。 |

### 事务失效的场景

1. 数据库引擎不支持事务，比如mysql数据库使用的是MyISAM引擎；
2. 事务注解所在的方法是非public的，因为SpringAOP默认使用CGLIB代理，无法给非public的方法创建代理，导致事务切面无法切入；



1. 异常类型不正确或被捕获，默认只回滚运行时异常和error，受检异常（IO异常，SQL异常）被视为业务异常，默认会提交事务；另外事务代理会通过捕获目标方法抛出的异常来决定是回滚还是提交，如果异常在方法内部被catch捕获且没有被重新抛出，代理会认为方法执行成功，从而提交事务；
2. 一个没有事务注解的方法调用了同一个类中有注解的方法，这种情况下使用的是this对象，是真实的对象，而不是代理对象；
3. 错误配置了事务传播属性，比如配置成以非事务方式运行，事务也会不生效；
4. 试图在final或static方法上使用事务，CGLIB代理是通过生成目标类的子类来实现的，它无法重写final和static方法。

## Spring设计模式

### 工厂模式

Spring使用工厂模式可以通过BeanFactory或ApplicationContext创建bean对象。

**两者对比：**

BeanFactory：延迟注入(使用到某个bean的时候才会注入),相比于ApplicationContext来说会占用更少的内存，程序启动速度更快。

ApplicationContext：容器启动的时候，不管你用没用到，一次性创建所有bean。BeanFactory仅提供了最基本的依赖注入支持，ApplicationContext扩展了BeanFactory, 除了有BeanFactory的功能还有额外更多功能(比如：支持基于观察者模式的事件驱动编程，另外Java EE的标准注解，如@Resource也自动支持)，所以一般开发人员使用ApplicationContext会更多。

### 单例设计模式

在我们的系统中，有一些对象其实我们只需要一个，比如说：线程池、缓存、日志对象等。事实上，这一类对象只能有一个实例，如果制造出多个实例就可能会导致一些问题的产生，比如：资源使用过量、或者结果不一致性。

**使用单例模式的好处**:

对于频繁使用的对象，可以省略创建对象所花费的时间，这对于那些大对象而言，是非常可观的一笔系统开销；

由于new操作的次数减少，因而对系统内存的使用频率也会降低，这将减轻GC压力，缩短GC停顿时间。

**Spring中bean的默认作用域就是singleton(单例)的。**除了singleton作用域，Spring中bean还有下面几种作用域：

* **prototype**:每次获取都会创建一个新的bean实例。也就是说，连续getBean()两次，得到的是不同的Bean实例。
* **request**（仅Web应用可用）:每一次HTTP请求都会产生一个新的bean（请求bean），该bean仅在当前HTTP request内有效。
* **session**（仅Web应用可用）:每一次来自新session的HTTP请求都会产生一个新的bean（会话bean），该bean仅在当前HTTP session内有效。
* **global-session**（仅Web应用可用）：每个Web应用在启动时创建一个Bean（应用Bean），该bean仅在当前应用启动时间内有效。

### 代理模式

**一个经典的例子就是AOP，**能够将那些与业务无关，却为业务模块所共同调用的逻辑（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度。

Spring AOP就是基于动态代理的，如果要代理的对象，实现了某个接口，那么Spring AOP会使用**JDK**去创建代理对象，而对于没有实现接口的对象，Spring AOP会使用**Cglib**生成一个被代理对象的子类来作为代理。

### 观察者模式

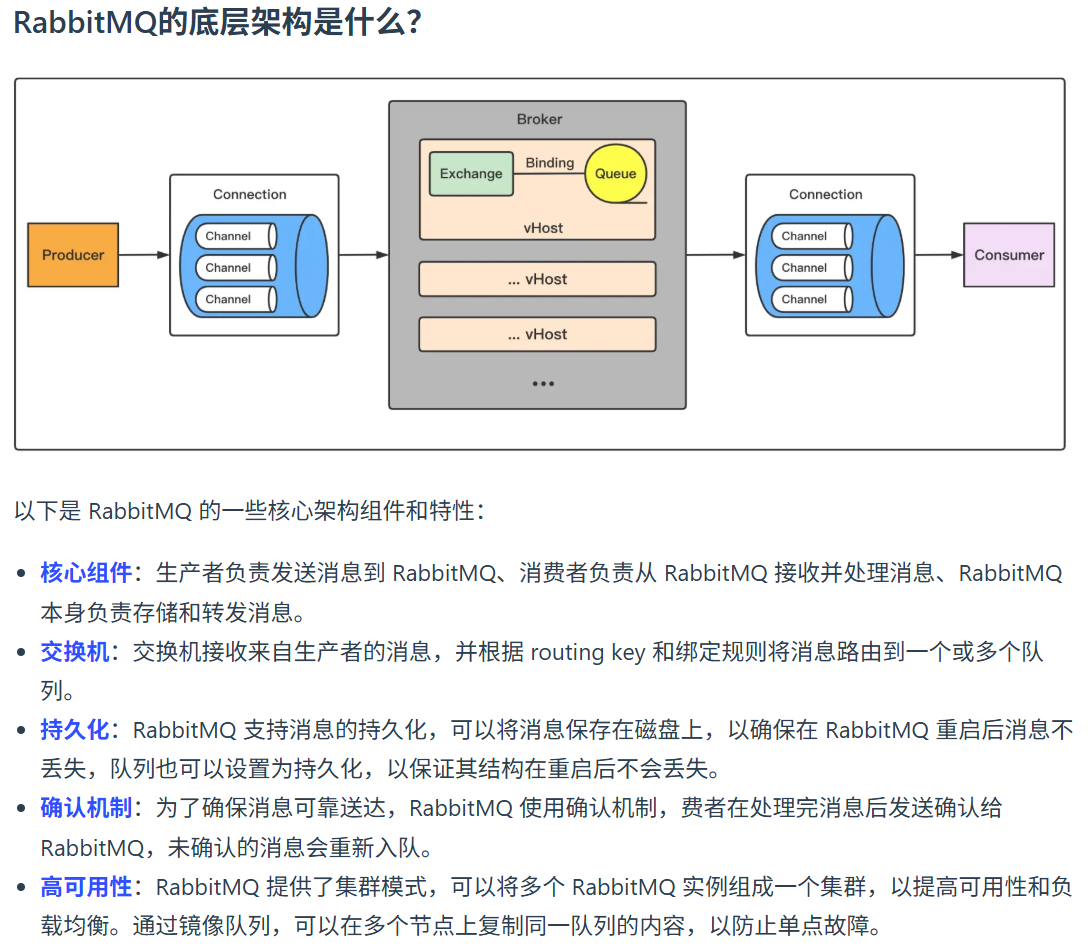
观察者模式表示的是一种对象与对象之间具有依赖关系，当一个对象发生改变的时候，依赖这个对象的所有对象也会做出反应。Spring事件驱动模型就是观察者模式很经典的一个应用。比如我们每次添加商品的时候都需要重新更新商品索引，这个时候就可以利用观察者模式来解决这个问题。

### 适配器模式

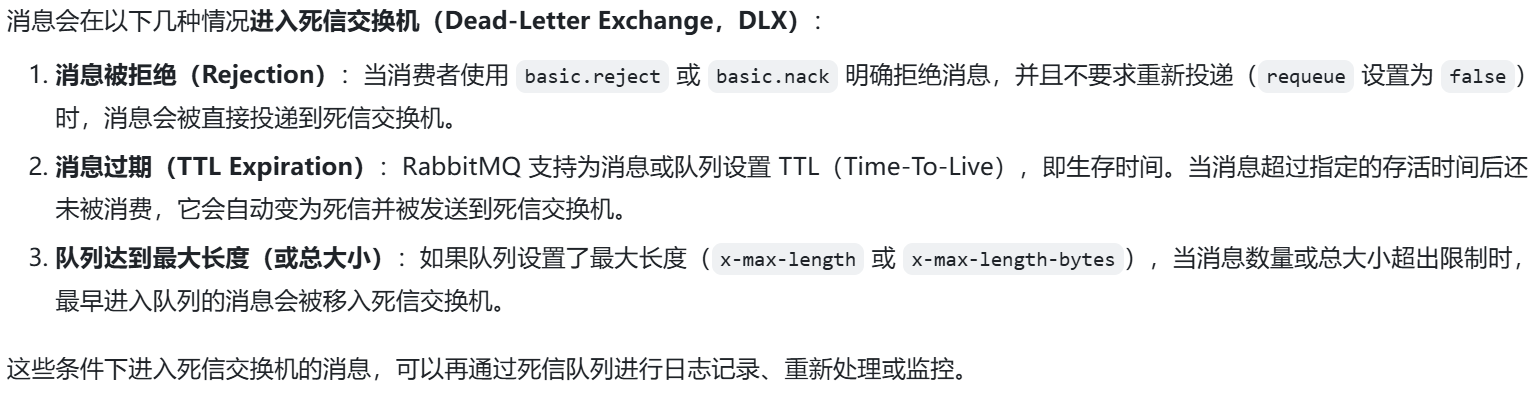
适配器模式(Adapter Pattern)可以将一个接口转换成我们希望的另一个接口，适配器模式使接口不兼容的那些类可以一起工作。

# RabbitMQ部分

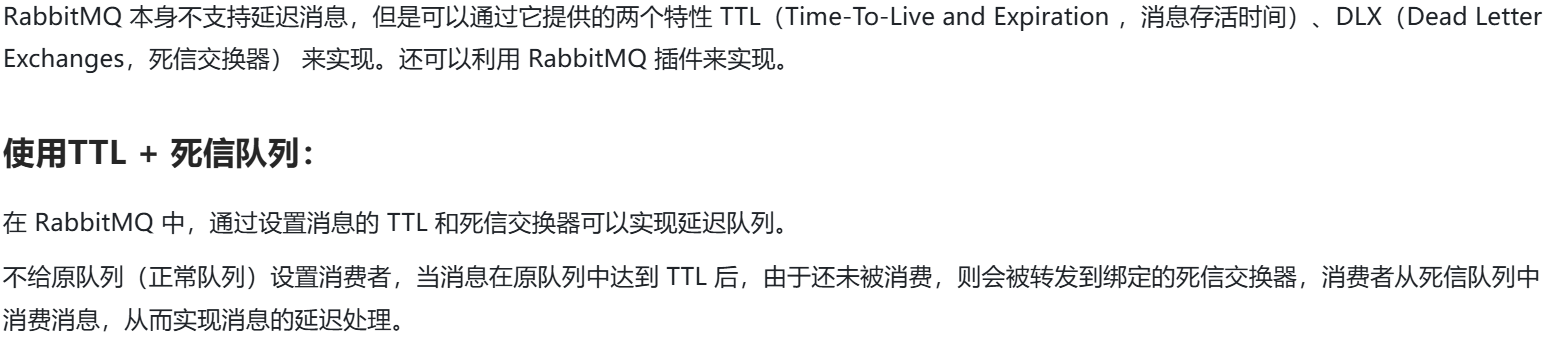
## RabbitMQ的底层架构



## 消息什么情况下会进入死信队列？



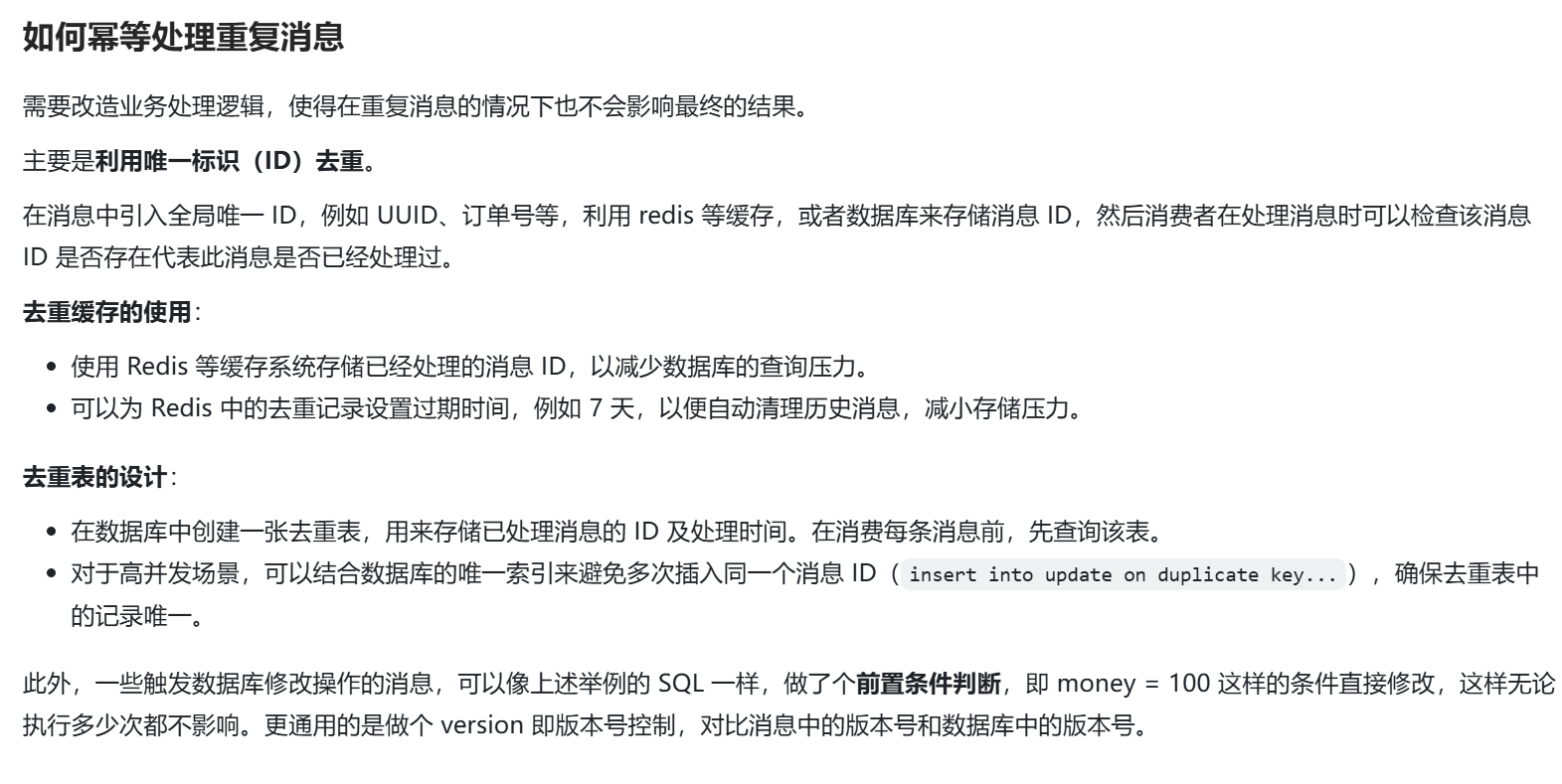
## RabbitMQ怎样实现延迟队列？



## RabbitMQ中无法路由的消息会去哪里？



## 如何避免重复处理消息？



## RabbitMQ的推和拉模式？

推模式：

推模式也称为订阅模式。消息是主动推送给消费者的，消费者会预先注册一个回调函数（消费者处理器），消息到达队列后会立刻推送给消费者，消费者设置预取数量来控制流量。这样的话消息的实时性高，到达后立即推送，减少了不必要的请求开销；缺点的话可能因处理能力不足导致消息堆积，需要合理设置预取数量以避免过载。适用于实时性要求高，消息量稳定的场景。

拉模式：

拉模式需要消费者主动从队列中获取消息。消费者主动请求，可以精确控制获取消息的时机和数量，适用于消息处理耗时较长或需要批量处理的场景。消费者可以按照自身能力获取消息，避免消息积压在消费者端，适合处理耗时任务。但是实时性较差，需要轮询去获取消息，大量的获取请求浪费可能增加了网络请求开销。适用于处理耗时，需要精确控制的场景。

## 消费者消费失败的常见因素？

（1）消息处理逻辑错误：比如消息内容是订单支付成功通知，但是消费者处理时因逻辑错误（比如金额计算错误）导致异常；也可能是外部依赖的异常，比如消息需要调用第三方的API，但是接口返回超时或者错误响应

（2）消费者也可能因为服务突然宕机，导致正在处理的消息未确认；或者消费者处理消息时执行耗时操作（比如生成大型图表），长时间未返回ACK触发RabbitMQ的超时机制。

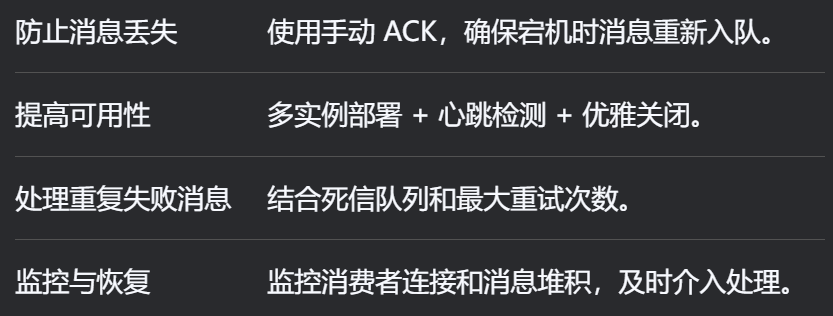
（3）网络或中间件问题：消费者与RabbitMQ之间的网络抖动，导致心跳超时或通道（Channel）关闭；或者集群中某个节点宕机，消费者未正确切换到其他节点，导致消息无法投递。

## 消费者怎么保证消息消费的可靠性？

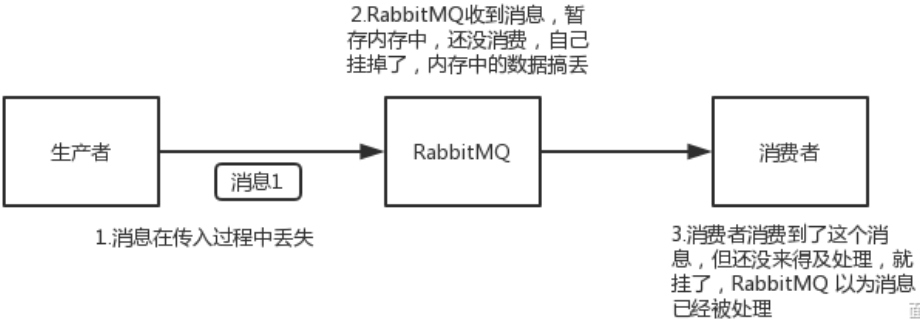


## 如果消费者宕机要怎么办？

首先可以通过心跳检测检查故障的节点，然后因为宕机前未发生ACK，消息未确认，所以可以等消费者重新连接之后重新入队处理。



## RabbitMQ消息丟失的3种情况和对应解决？



（1）生产者弄丢数据

生产者将数据发送到RabbitMQ的时候，可能数据在半路弄丢了（比如网络问题之类的）。RabbitMQ有两种解决方式：1、生产者发送数据之前开启RabbitMQ的**事务功能**，如果消息没有成功被MQ接收到，那么生产者就会收到异常报错，此时就可以回滚事务，然后**重试发送消息**，但是问题是事务机制是同步的，提交一个事务之后就会阻塞等待事务处理完，所以吞吐量会下降，比较消耗性能；

2、开启MQ的**confirm模式**，这样每次写的消息都会分配一个唯一的id，然后如果写入了RabbitMQ中，MQ就会回传一个ack消息，告诉生产者这个消息ok了，如果MQ没能处理这个消息就会回调生产者的nack接口，告诉生产者消息接收失败了，然后可以进行重发。这个过程是异步的，发送完这个消息之后可以紧接着发生下一个消息。

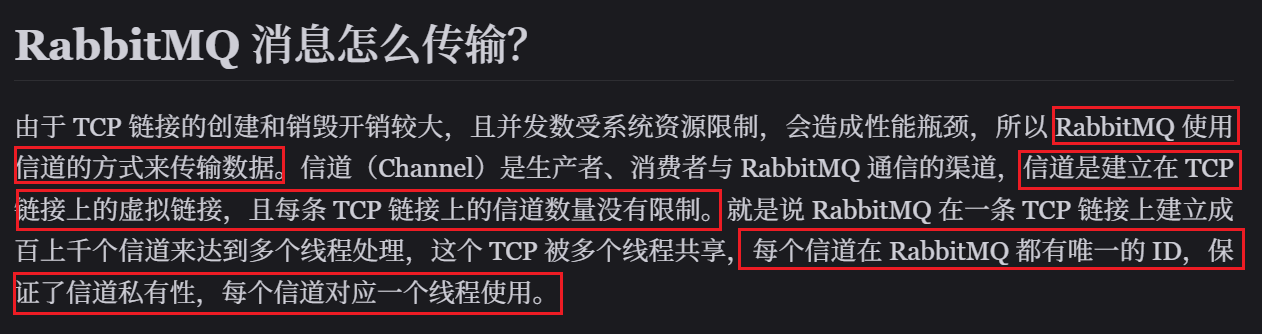
（2）RabbitMQ弄丢了数据

这个我们需要开启RabbitMQ的持久化，保证消息写入之后会**持久化**到磁盘，这样即使RabbitMQ自己挂了，恢复之后也可以自动读取之前存储的数据。（需要在创建队列的时候给queue设置为持久化，并且还需要给消息也设置为持久化的）

（3）消费者弄丢了数据

比如消息刚收到消费者就宕机了，这种情况我们需要**关闭MQ的默认ack机制**（因为默认是收到了就会触发ack确认），我们应该在消费者这边**业务处理完成之后再手动去ack确认**，这样即使消费者宕机没能处理消息，消息也不会被ack，所以消息可以重新入队处理。

## RabbitMQ消息怎么传输？



## RabbitMQ怎么保证消息的顺序性？

**方案一：单队列单消费者（FIFO最简单模式）**

这是最直接但也限制最大的方法。

**原理**：

* **生产者**：将需要保证顺序的所有消息发送到**同一个队列**中。
* **消费者**：该队列**只能有一个消费者**，并且设置channel.basicQos(1)，即每次只取一条消息，处理完一条再取下一条。

**优点**：

* 实现简单，充分利用了RabbitMQ队列本身的FIFO（先进先出）特性。

**缺点**：

* **无法水平扩展**：单个消费者是性能瓶颈，吞吐量低。
* **单点风险**：如果该消费者宕机，虽然消息不会丢，但处理会完全停止。
* **适用场景**：消息量非常小，对吞吐量要求不高的场景。

**方案二：根据消息ID或业务键进行分组（路由到同队列）**

这是最常用且合理的方案，核心思想是：**将需要保证顺序的消息通过路由键确保它们进入同一个队列，并被同一个消费者顺序处理**。

**原理**：

* **消息分组**：对消息进行分组。例如，订单ID为order\_123的所有消息（创建、付款、发货）必须顺序处理。
* **生产端**：使用**一致性哈希交换器**或**自定义路由键**，将同一组的消息总是路由到同一个队列。

**例如**：使用x-modulus-hash这类交换器，以订单ID order\_123作为路由键，计算出的哈希值总会将其路由到队列queue\_2。

* **消费端**：为**每个队列启动一个消费者**（可以是多个队列，即多个消费者实例，每个实例处理不同组的消息）。同样，每个消费者需要设置prefetchCount=1。

**工作流程**：

订单A（ID=1）的所有消息 → 路由键order\_1→始终发往**队列1**→ 由**消费者1**处理。

* 订单B（ID=2）的所有消息 → 路由键order\_2→ 始终发往**队列2**→ 由**消费者2**处理。

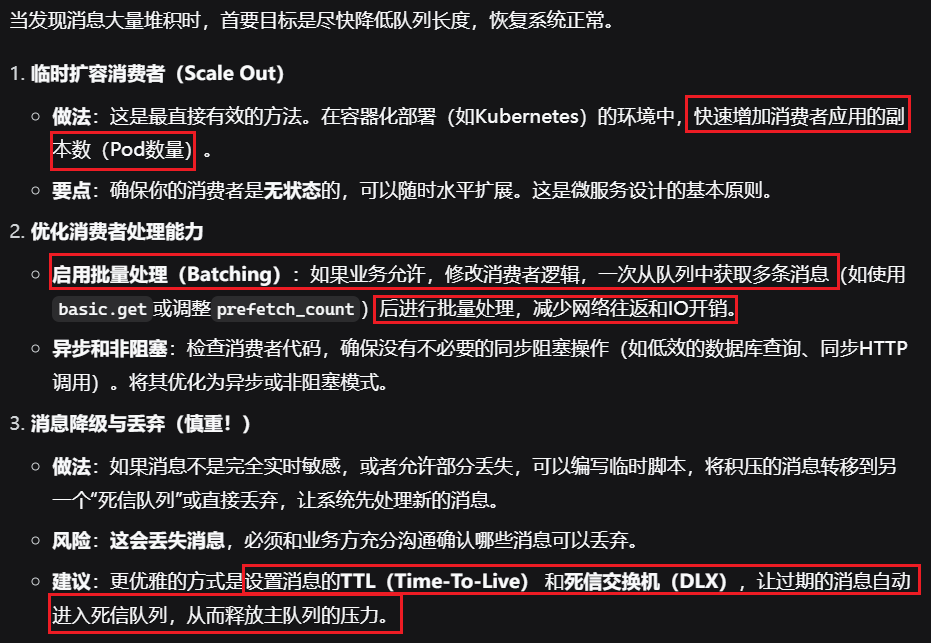
**优点**：

* **高性能且可扩展**：不同的组（如不同的订单）可以被不同的消费者并行处理，解决了方案一的瓶颈问题。
* **逻辑清晰**：符合大部分业务场景（如订单、会话跟踪）。

**缺点**：

* 需要提前规划好消息的分组逻辑。
* 如果某个组的消息特别多（“热点订单”），对应的队列和消费者可能会有压力，但通常这种情况较少。

## RabbitMQ消息堆积怎么处理





## 优惠劵场景，RabbitMQ保证一致性问题

本地消息表的核心思想是**将分布式事务拆分为两个本地事务**，通过数据库的事务特性和重试机制保证最终一致性：

第一个本地事务：在业务数据库中同时完成订单状态更新和消息记录

第二个本地事务：通过定时任务将记录的消息可靠地发送到MQ

**具体实现流程：**

**第一阶段：业务处理与消息记录**

**开启数据库事务**：当优惠券被抢后，系统开始处理优惠券状态更新

**更新订单状态**：在同一个数据库事务中，将优惠券状态从"待发放"改为"已发放"

**写入本地消息表**：在同一事务中，向专门设计的消息表插入一条记录，包含：

消息ID（唯一标识）、消息内容（JSON格式的优惠券数据）、消息状态（初始为"待发送"）、创建时间、重试次数（初始为0）

**提交事务**：只有当优惠券更新和消息记录都成功后才提交事务

**第二阶段：消息发送与状态更新**

**定时任务扫描**：系统有一个独立的定时任务，定期（如每5秒）扫描本地消息表中状态为"待发送"的记录

**发送MQ消息**：对于每条待发送记录：尝试将消息内容发送到消息队列。如果发送成功：更新该消息记录状态为"已发送"；如果发送失败：增加重试次数、记录错误日志、保持状态为"待发送"

**重试机制**：对于发送失败的消息，定时任务会在下次扫描时重新尝试发送，直到：发送成功或达到最大重试次数（如5次），此时可将状态改为"发送失败"并报警。

# 操作系统部分

## 进程之间的通讯方式

* **匿名管道：**用于**具有亲缘关系**的进程之间的通信，比如父子进程、兄弟进程，遵循**先进先出**原则，数据一旦被读出，便不在管道中存在，**不可反复读取**；容量有限，如果管道满了，写操作会阻塞；如果管道空了，读操作会阻塞；
* **命名管道：突破了亲缘关系的限制**，任何无关进程都可以通过访问该路径进行通信，其他和匿名管道一样；
* **消息队列：支持无关进程间通信**。消息有类型，读进程**可以根据类型有选择地接收消息**，而不必须是FIFO顺序。内核中的消息队列有最大长度和最大消息数的限制。
* **共享内存：**允许多个进程共享同一块物理内存区域。进程通过系统调用创建一块共享内存区，并**将其映射到自己的虚拟地址空间**。之后，进程就可以像访问普通内存一样读写这块区域。**速度极快**：因为数据不需要在内核和用户空间之间来回拷贝。**需要同步机制**（如信号量或互斥锁）：因为多个进程同时读写同一块内存，会产生竞争，必须由程序员自己控制同步。
* **信号量：本质不是用来传递数据的**，而是一个**计数器**，用于实现进程间的**同步与互斥**（防止多个进程同时访问一个共享资源）。它主要用于控制多个进程对共享资源（如共享内存）的访问。信号量的值表示可用资源的数量。通常与共享内存等方式配合使用，解决其同步问题。
* **信号：异步通信机制**，用于通知接收进程某个事件已经发生。进程可以发送信号给自身或其他进程，内核也可以在特定事件（如除零错误、用户按下Ctrl+C）发生时发送信号给进程。特点是**开销小，但能携带的信息量极少**（只有一个信号编号）。用于处理**异常、中断或简单的进程控制**（如终止、暂停、继续）。
* **套接字socket：**不仅可以用于同一台主机的进程间通信，更主要用于**不同主机**（网络）上的进程间通信。套接字是一种通信端点，通过IP地址、端口号和协议类型来定义。它支持多种协议，最常见的是TCP（面向连接、可靠）和UDP（无连接、不可靠）。适用范围最广（跨网络）。功能强大但设置相对复杂。