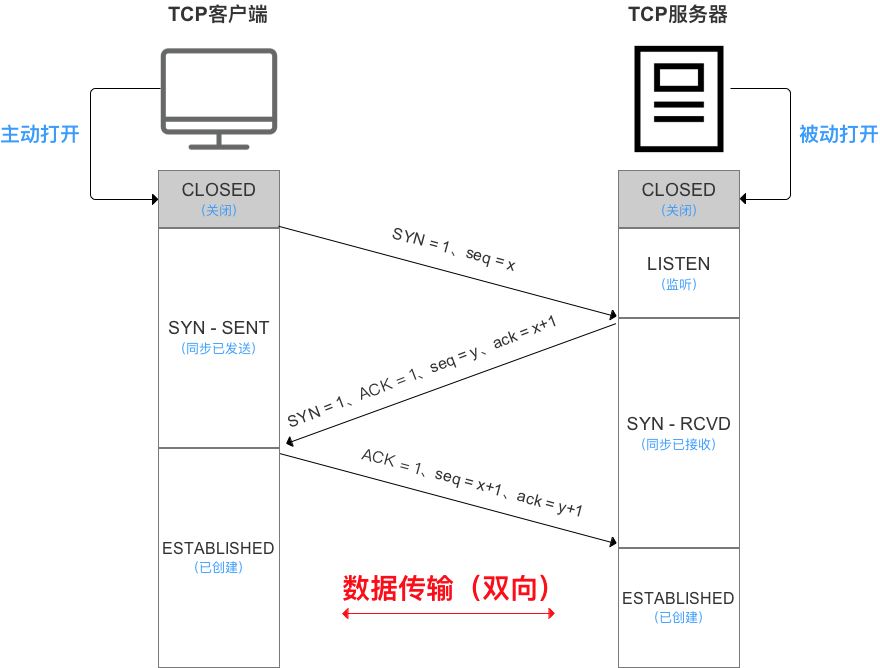
Java基础知识手册

## HTTP协议：

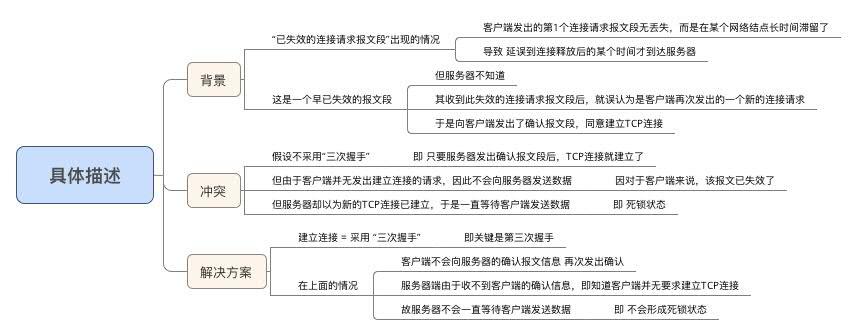
## TCP

### 建立连接与释放连接

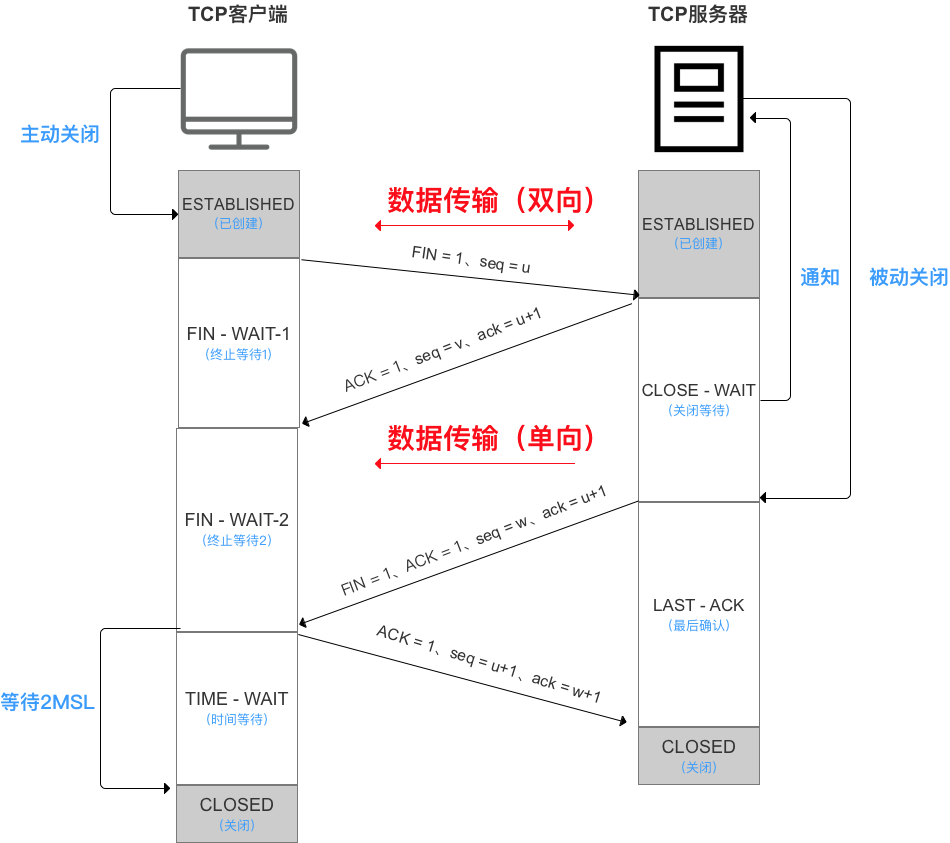




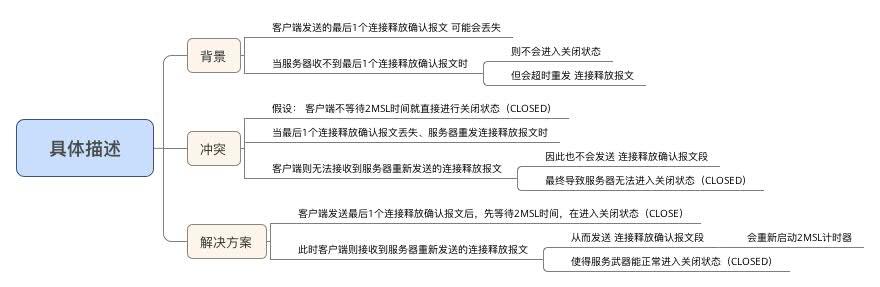
为什么三次握手：



从上可看出：服务端的TCP资源分配时刻 = 完成第二次握手时；而客户端的TCP资源分配时刻 = 完成第三次握手时







原因2：防止 上文提到的早已失效的连接请求报文 出现在本连接中  
客户端发送了最后1个连接释放请求确认报文后，再经过2MSL时间，则可使本连接持续时间内所产生的所有报文段都从网络中消失。

考虑这种情况，服务器运行在80端口，客户端使用的连接端口是12306，数据传输完毕后服务端主动关闭连接，但是没有进入TIME\_WAIT，而是直接计入CLOSED了。这时，客户端又通过同样的端口12306与服务端建立了一个新的连接。假如上一个连接过程中网络出现了异常，导致了某个包重传并延时到达了服务端，这时服务端就无法区分这个包是上一个连接的还是这个连接的。所以，主动关闭连接一方要等待2MSL，然后才能CLOSE，保证连接中的IP包都要么传输完成，要么被丢弃了。

### 无差错传输

每收到一个确认帧，发送窗口就向前滑动一个帧的距离

当发送窗口内无可发送的帧时（即窗口内的帧全部是已发送但未收到确认的帧），发送方就会停止发送，直到收到接收方发送的确认帧使窗口移动，窗口内有可以发送的帧，之后才开始继续发送

对于接收端：当收到数据帧后，将窗口向前移动一个位置，并发回确认帧，若收到的数据帧落在接收窗口之外，则一律丢弃。



只有接收窗口向前滑动、接收方发送了确认帧时，发送窗口才有可能（只有发送方收到确认帧才是一定）向前滑动

停止-等待协议、后退N帧协议 & 选择重传协议只是在发送窗口大小和接收窗口大小上有所差别：

停止等待协议：发送窗口大小=1，接收窗口大小=1；即 单帧滑动窗口 等于 停止-等待协议

后退N帧协议：发送窗口大小>1，接收窗口大小=1。

选择重传协议：发送窗口大小>1，接收窗口大小>1。

当接收窗口的大小为1时，可保证帧有序接收。

数据链路层的滑动窗口协议中，窗口的大小在传输过程中是固定的（注意要与TCP的滑动窗口协议区别）

实现无差错传输的解决方案

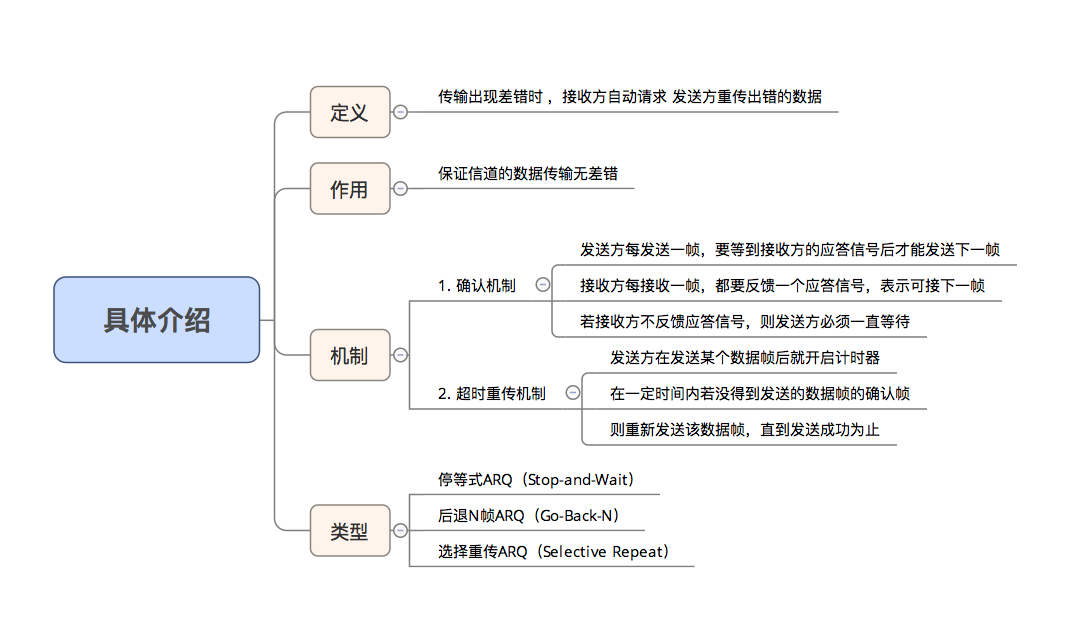
核心思想：采用一些可靠传输协议，使得出现差错时，让发送方重传差错数据：即

出错重传

当接收方来不及接收收到的数据时，可通知发送方降低发送数据的效率：即

速度匹配

针对上述2个问题，分别采用的解决方案是：自动重传协议 和 流量控制 & 拥塞控制协议



#### 出错重传

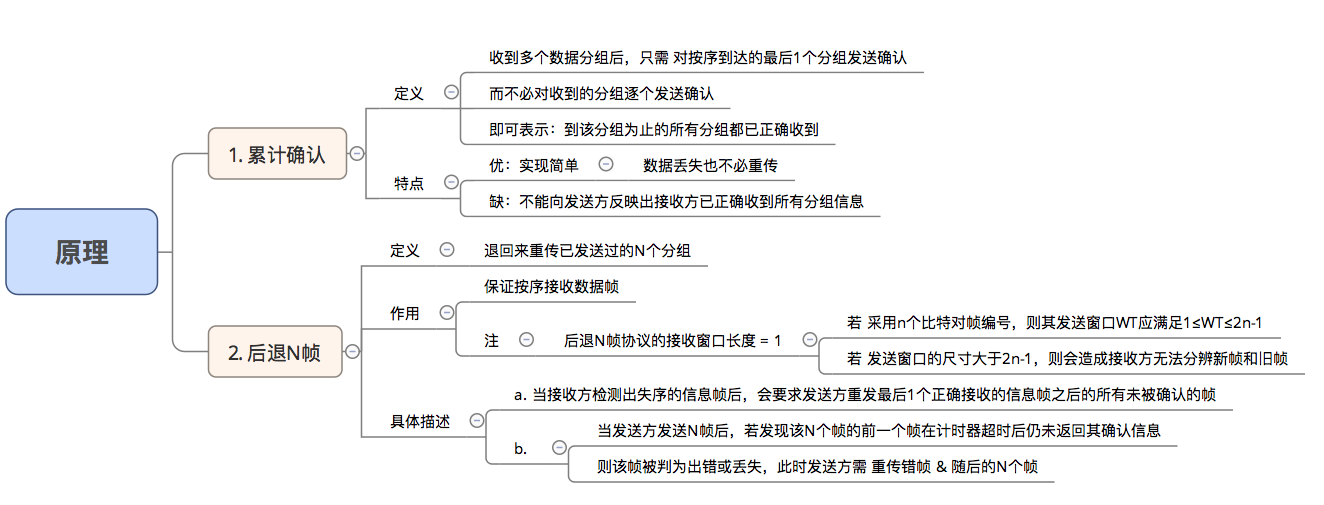
退N帧协议

连续ARQ协议

原理  
多帧滑动窗口 + 累计确认 + 后退N帧 + 超时重传

即 ：发送窗口大小>1、接收窗口大小=1

具体描述  
a. 发送方：采用多帧滑动窗口的原理，可连续发送多个数据帧 而不需等待对方确认  
b. 接收方：采用 **累计确认 & 后退N帧**的原理，只允许按顺序接收帧。具体原理如下：



**选择重传ARQ**

原理  
多帧滑动窗口 + 累计确认 + 后退N帧 + 超时重传

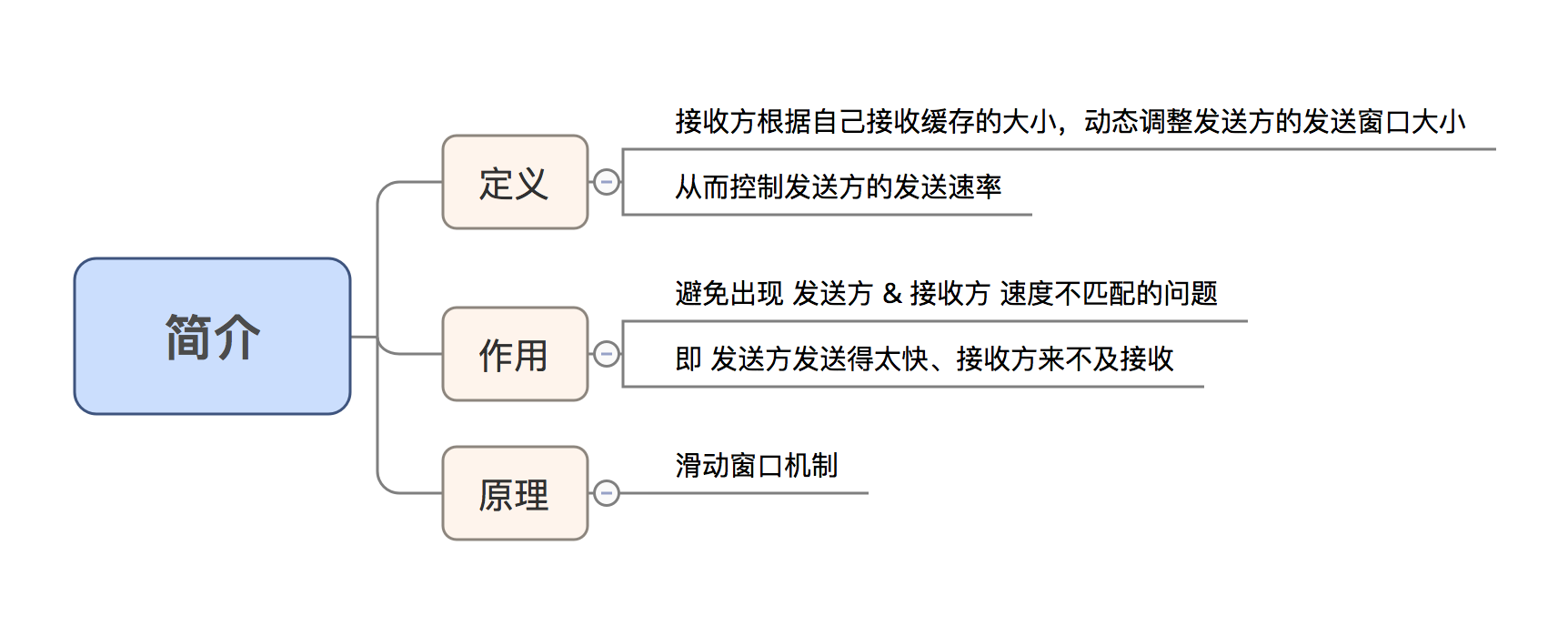
即 ：发送窗口大小>1、接收窗口大小>1

类似于类型2（后退N帧协议），此处仅仅是接收窗口大小的区别，故此处不作过多描述

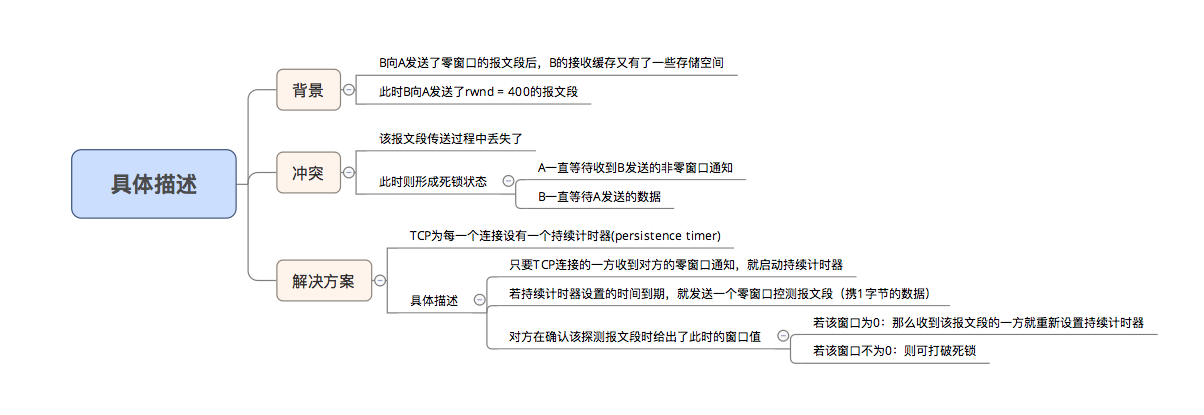
特点  
a. 优：因连续发送数据帧而提高了信道的利用率  
b. 缺：重传时又必须把原来已经传送正确的数据帧进行重传（仅因为这些数据帧前面有一个数据帧出了错），将导致传送效率降低

由此可见，若信道传输质量很差，导致误码率较大时，后退N帧协议不一定优于停止-等待协议

流量控制 & 拥塞控制



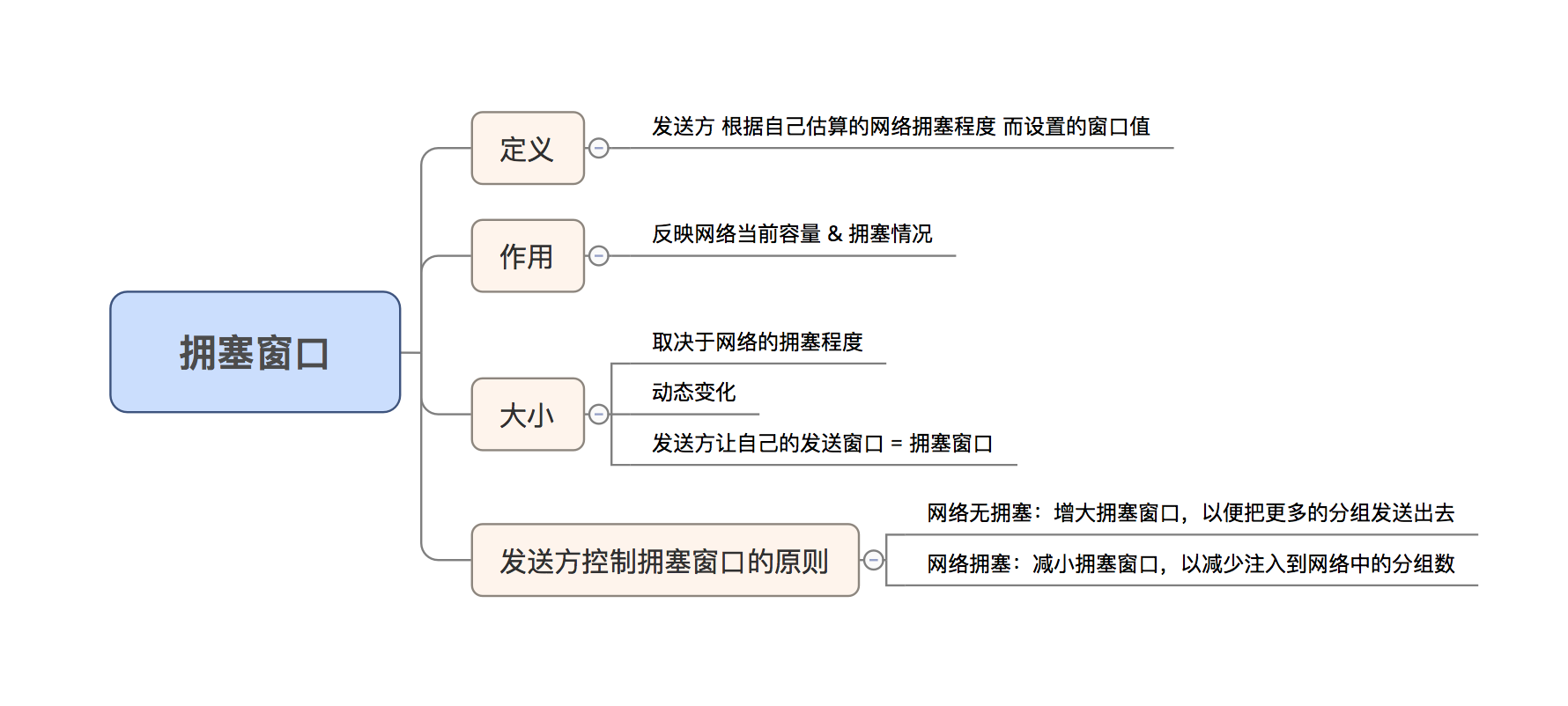
注意死锁问题



#### 拥塞控制

慢开始 & 拥塞避免

拥塞窗口



**慢开始算法**

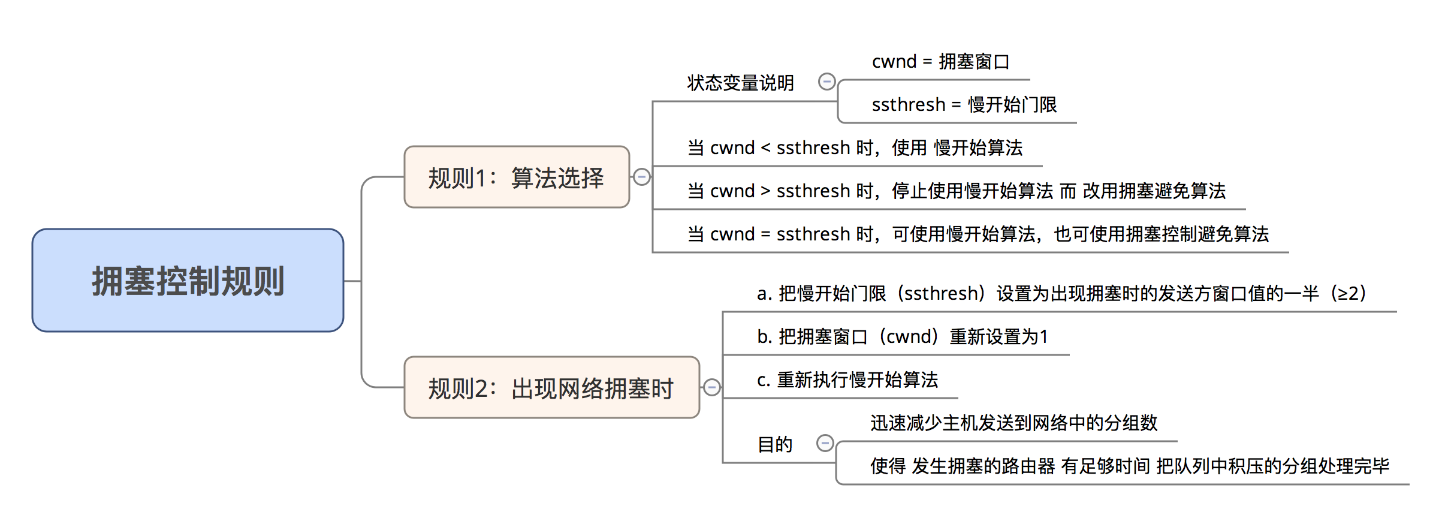
* 原理  
  当主机开始发送数据时，由小到大逐渐增大 拥塞窗口数值（即 发送窗口数值），从而 由小到大逐渐增大发送报文段，一般是两倍
* 目的  
  开始传输时，**试探**网络的拥塞情况

拥塞避免 算法

原理  
使得拥塞窗口（cwnd）**按线性规律 缓慢增长**：每经过一个往返时间RTT，发送方的拥塞窗口（cwnd）加1

**拥塞避免 并不可避免拥塞**，只是将拥塞窗口按现行规律缓慢增长，使得网络比较不容易出现拥塞

相比慢开始算法的加倍，拥塞窗口增长速率缓慢得多



**快重传 & 快恢复**

a. 快重传算法

* 原理

接收方 每收到一个失序的报文段后 就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方），而不要等到自己发送数据时才进行捎带确认

发送方只要一连收到3个重复确认就立即重传对方尚未收到的报文段，而不必 继续等待设置的重传计时器到期

* 作用  
  由于发送方尽早重传未被确认的报文段，因此采用快重传后可以使整个网络吞吐量提高约20%

### b. 快恢复

当发送方连续收到3个重复确认后，就：

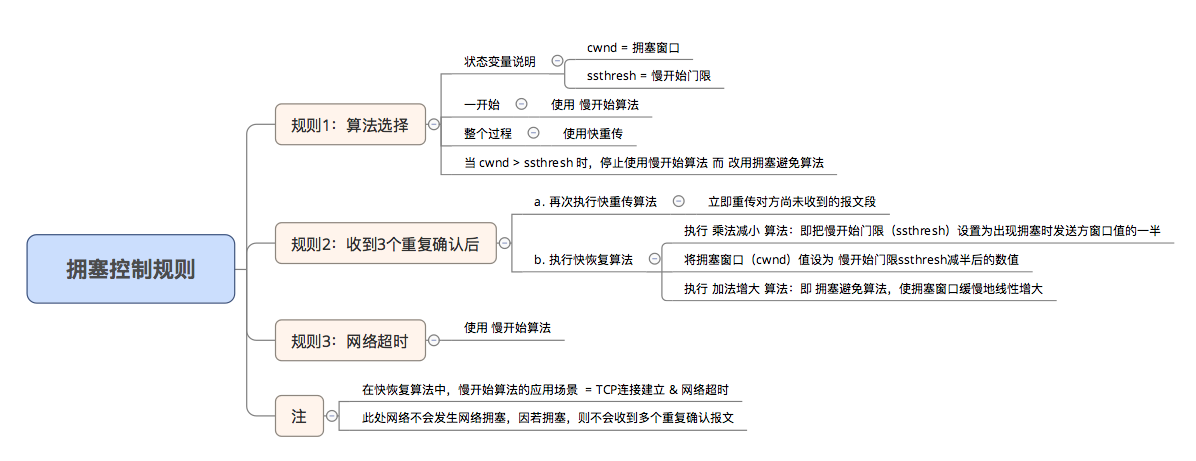
执行 **乘法减小** 算法：把 慢开始门限（ssthresh）设置为 出现拥塞时发送方窗口值的一半 = 拥塞窗口的1半

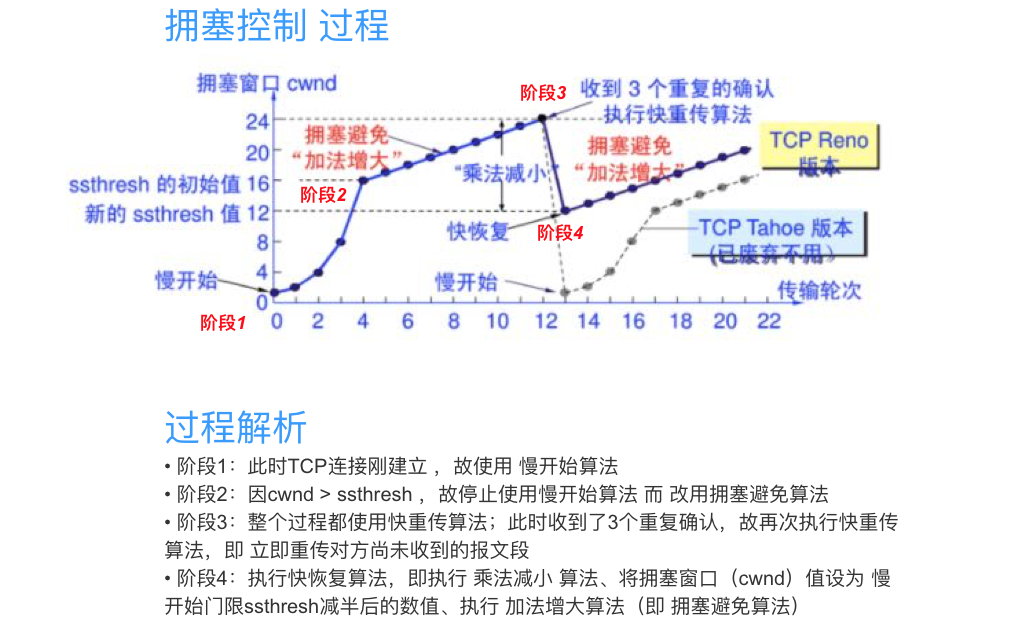
将拥塞窗口（cwnd）值设置为 慢开始门限ssthresh减半后的数值 = 拥塞窗口的1半

执行 **加法增大** 算法：执行拥塞避免算法，使拥塞窗口缓慢地线性增大。

由于跳过了拥塞窗口（cwnd）从1起始的慢开始过程，所以称为：快恢复

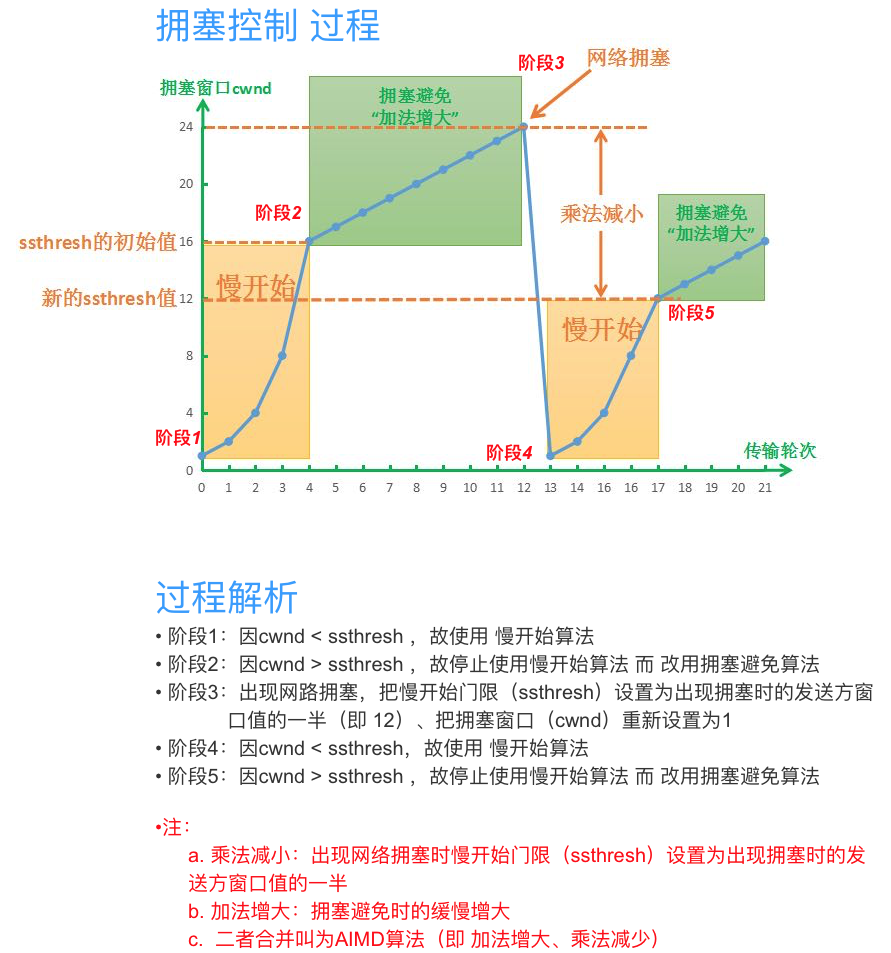
此处网络不会发生网络拥塞，因若拥塞，则不会收到多个重复确认报文





## TCP与UDP的区别





# Java泛型

Generics are a facility of [generic programming](https://en.wikipedia.org/wiki/Generic_programming) that were added to the [Java programming language](https://en.wikipedia.org/wiki/Java_(programming_language)) in 2004 within version [J2SE](https://en.wikipedia.org/wiki/Java_Platform,_Standard_Edition) 5.0. They were designed to extend Java's [type system](https://en.wikipedia.org/wiki/Type_system) to allow “a type or method to operate on objects of various types while providing compile-time type safety”[[1]](https://en.wikipedia.org/wiki/Generics_in_Java#cite_note-1).

# Java多线程

# Java 锁机制

[java并发系列：深入分析Synchronized](https://blog.csdn.net/bohu83/article/details/51141836)

## CAS

在JDK 5之前Java语言是靠synchronized关键字保证同步的，这会导致有锁

锁机制存在以下问题：

（1）在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。

（2）一个线程持有锁会导致其它所有需要此锁的线程挂起。

（3）如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。

volatile是不错的机制，但是volatile不能保证原子性。因此对于同步最终还是要回到锁机制上来。

独占锁是一种悲观锁，synchronized就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁用到的机制就是CAS，Compare and Swap。

一、什么是CAS

CAS,compare and swap的缩写，中文翻译成比较并交换。

我们都知道，在java语言之前，并发就已经广泛存在并在服务器领域得到了大量的应用。所以硬件厂商老早就在芯片中加入了大量直至并发操作的原语，从而在硬件层面提升效率。在intel的CPU中，使用cmpxchg指令。

在Java发展初期，java语言是不能够利用硬件提供的这些便利来提升系统的性能的。而随着java不断的发展,Java本地方法(JNI)的出现，使得java程序越过JVM直接调用本地方法提供了一种便捷的方式，因而java在并发的手段上也多了起来。而在Doug Lea提供的cucurenct包中，CAS理论是它实现整个java包的基石。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值 。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该 位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前 值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A，执行多步计算来获得新 值 B，然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改，则 CAS 操作成功。

类似于 CAS 的指令允许算法执行读-修改-写操作，而无需害怕其他线程同时 修改变量，因为如果其他线程修改变量，那么 CAS 会检测它（并失败），算法 可以对该操作重新计算。

二、CAS的目的

利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法。其它原子操作都是利用类似的特性完成的。而整个J.U.C都是建立在CAS之上的，因此对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。

三、CAS存在的问题

CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作

1.  ABA问题。因为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，变成了B，又变成了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。ABA问题的解决思路就是使用版本号。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的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

关于ABA问题参考文档: http://blog.hesey.net/2011/09/resolve-aba-by-atomicstampedreference.html

2. 循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

3. 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，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。

## Synchronized

类锁

对象锁

修饰静态方法锁 等同于类锁

synchronized关键字最主要有以下3种应用方式，下面分别介绍

* 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁
* 修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁
* 修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

实现原理：

修饰块

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

修饰对象

方法的同步并没有通过指令monitorenter和monitorexit来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了ACC\_SYNCHRONIZED标示符。JVM就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。 其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。



实例变量：存放类的属性数据信息，包括父类的属性信息，如果是数组的实例部分还包括数组的长度，这部分内存按4字节对齐。

填充数据：由于虚拟机要求对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐，这点了解即可。

| * 虚拟机位数 | 头对象结构 | 说明 |
| --- | --- | --- |
| 32/64bit | Mark Word | 存储对象的hashCode、锁信息或分代年龄或GC标志等信息 |
| 32/64bit | Class Metadata Address | 类型指针指向对象的类元数据，JVM通过这个指针确定该对象是哪个类的实例。 |

其中Mark Word在默认情况下存储着对象的HashCode、分代年龄、锁标记位等以下是32位JVM的Mark Word默认存储结构

| 锁状态 | 25bit | 4bit | 1bit是否是偏向锁 | 2bit 锁标志位 |
| --- | --- | --- | --- | --- |
| 无锁状态 | 对象HashCode | 对象分代年龄 | 0 | 01 |



这里我们主要分析一下重量级锁也就是通常说synchronized的对象锁，锁标识位为10，其中指针指向的是monitor对象（也称为管程或监视器锁）的起始地址。每个对象都存在着一个 monitor 与之关联，对象与其 monitor 之间的关系有存在多种实现方式，如monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态。在Java虚拟机(HotSpot)中，monitor是由ObjectMonitor实现的，其主要数据结构如下（位于HotSpot虚拟机源码ObjectMonitor.hpp文件，C++实现的）