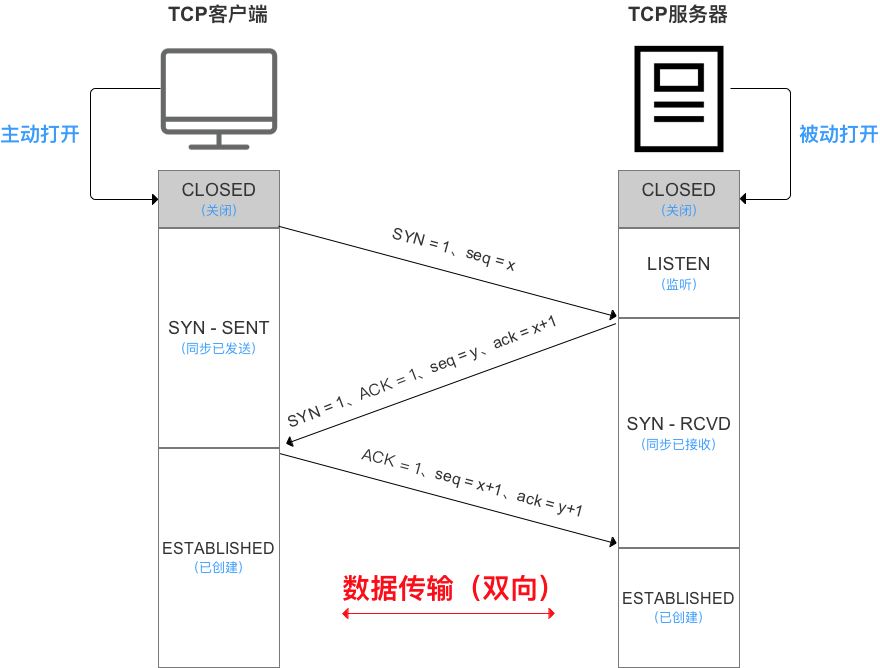
Java基础知识手册

## HTTP协议：

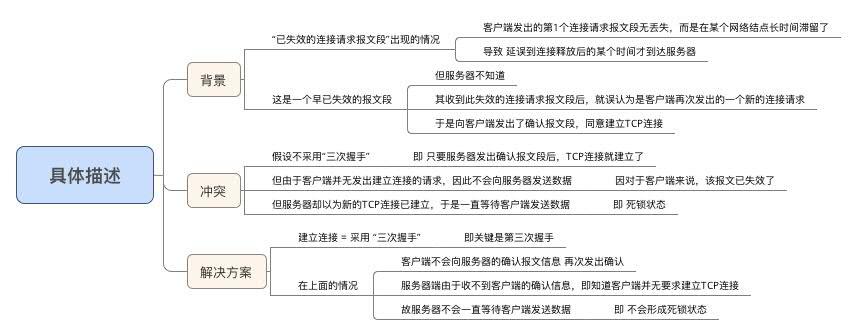
## TCP

### 建立连接与释放连接

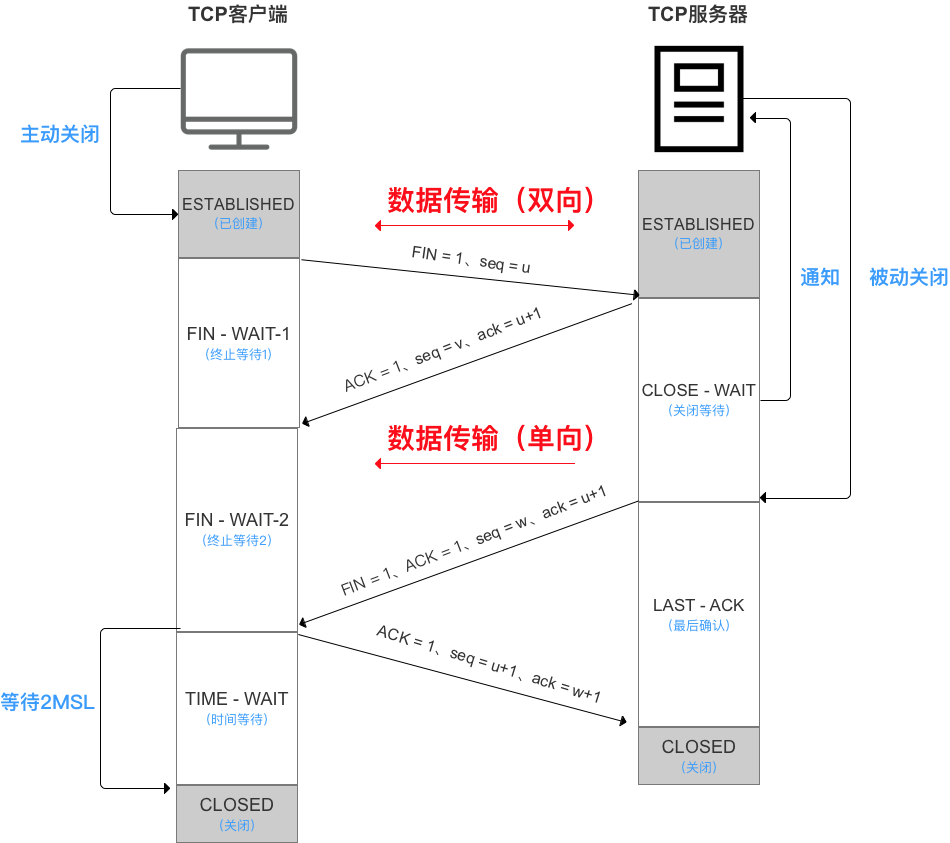




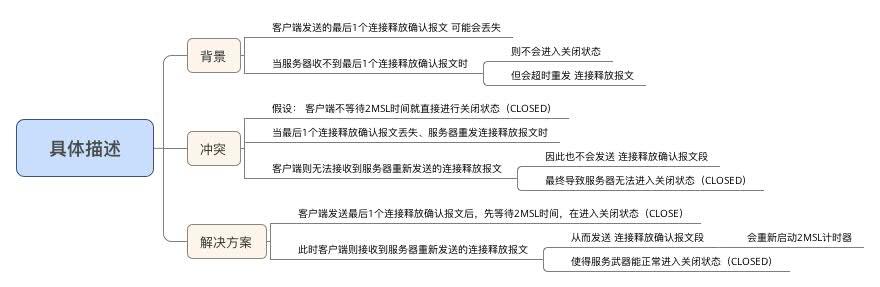
为什么三次握手：



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

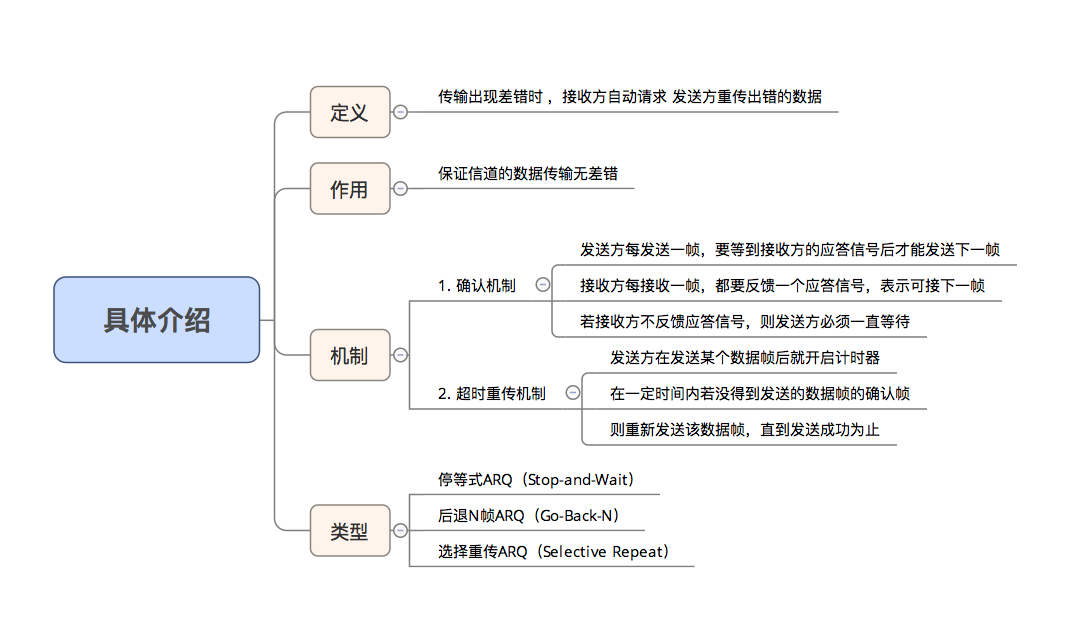
核心思想：采用一些可靠传输协议，使得出现差错时，让发送方重传差错数据：即

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

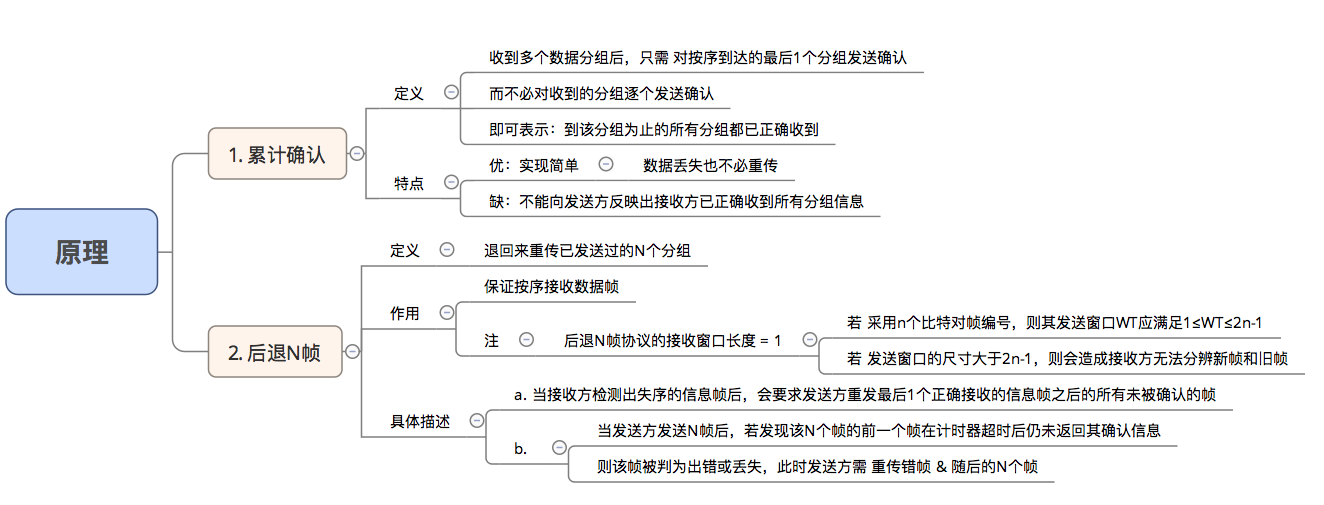
退N帧协议

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

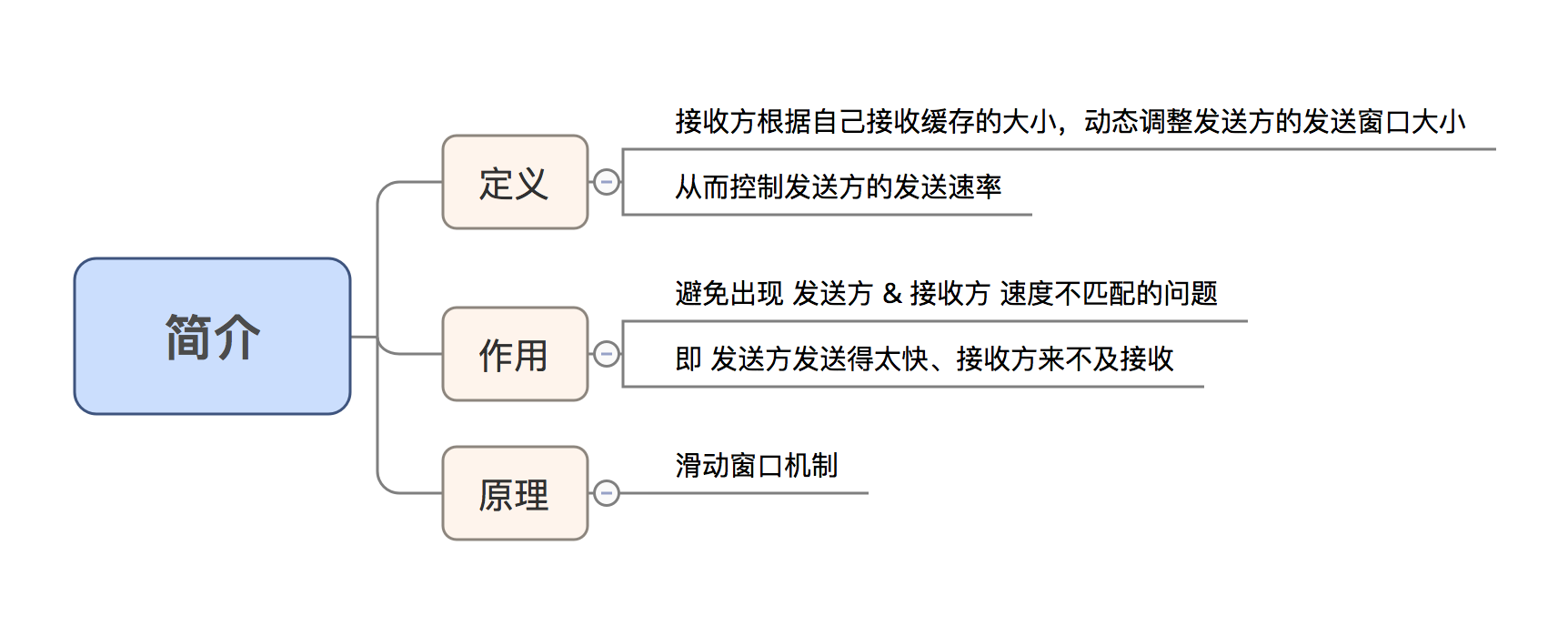
即 ：发送窗口大小>1、接收窗口大小>1

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

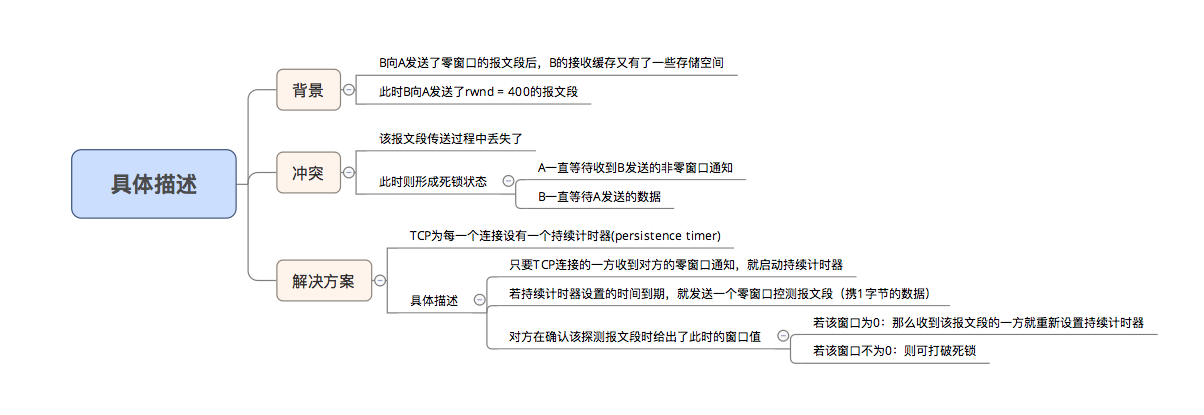
特点  
a. 优：因连续发送数据帧而提高了信道的利用率  
b. 缺：重传时又必须把原来已经传送正确的数据帧进行重传（仅因为这些数据帧前面有一个数据帧出了错），将导致传送效率降低

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



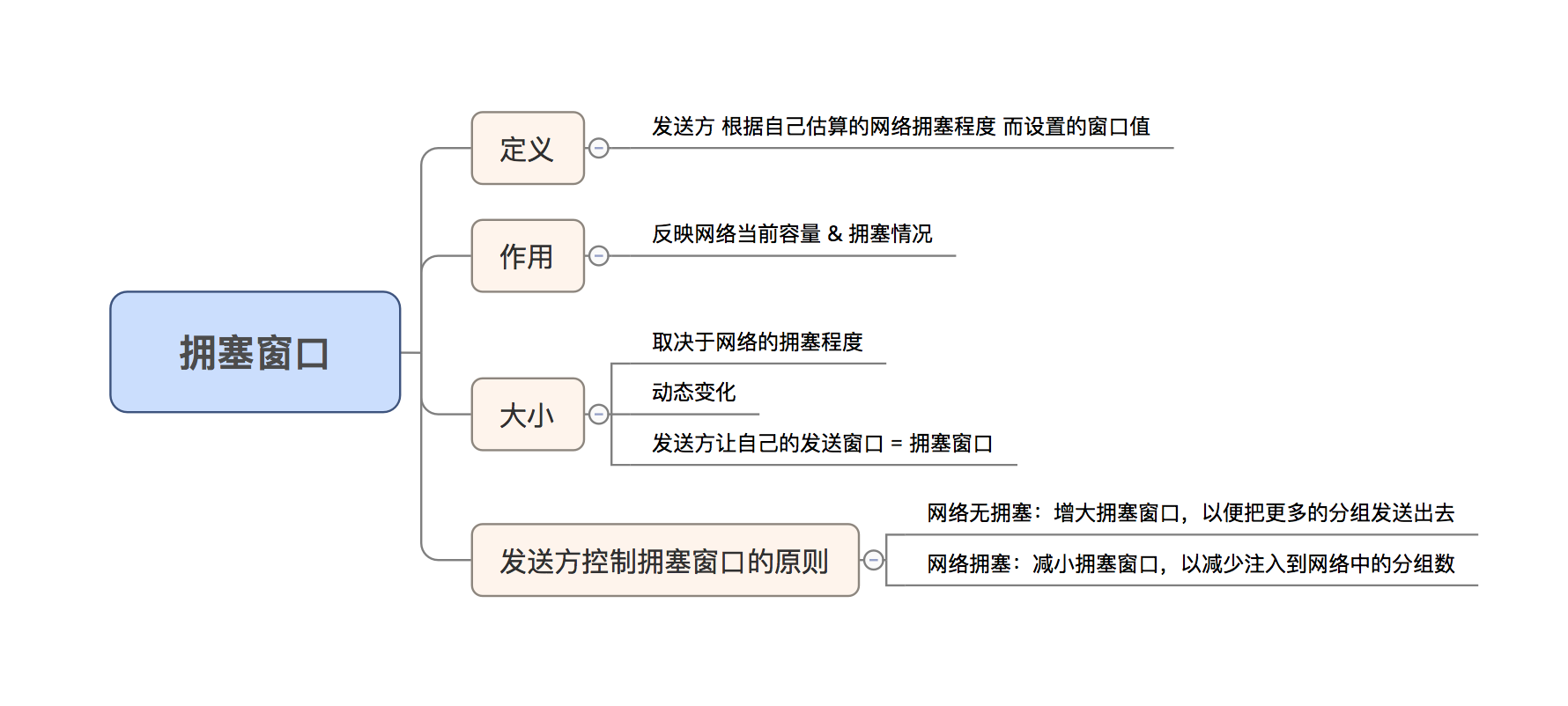
注意死锁问题



#### 拥塞控制

慢开始 & 拥塞避免

拥塞窗口



**慢开始算法**

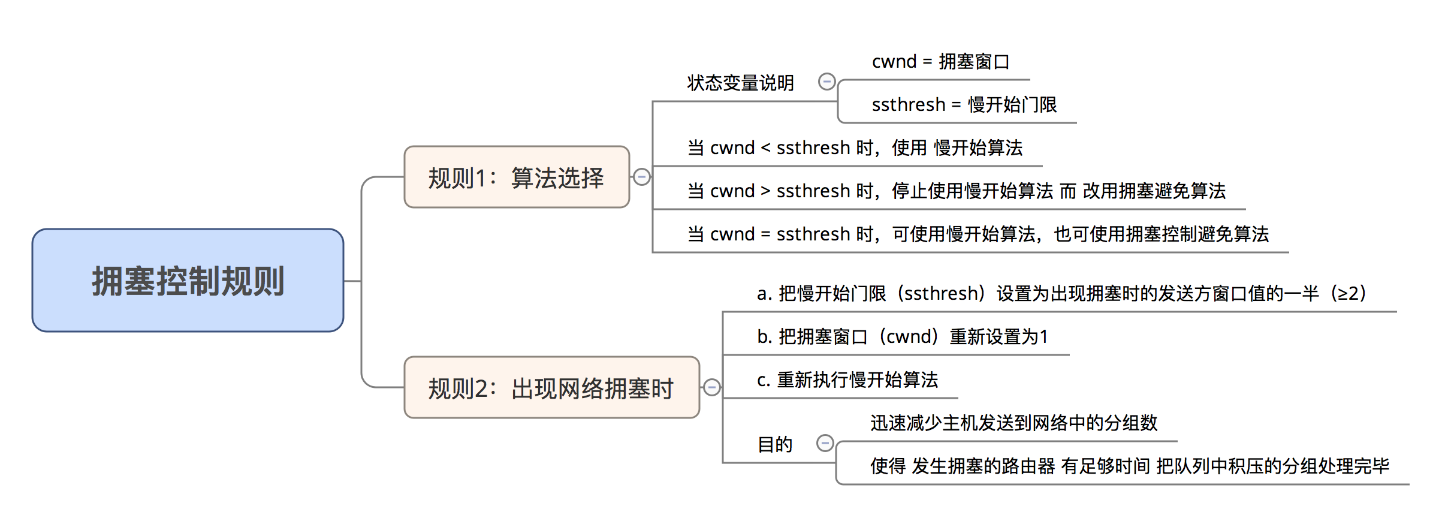
* 原理  
  当主机开始发送数据时，由小到大逐渐增大 拥塞窗口数值（即 发送窗口数值），从而 由小到大逐渐增大发送报文段，一般是两倍
* 目的  
  开始传输时，**试探**网络的拥塞情况

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：

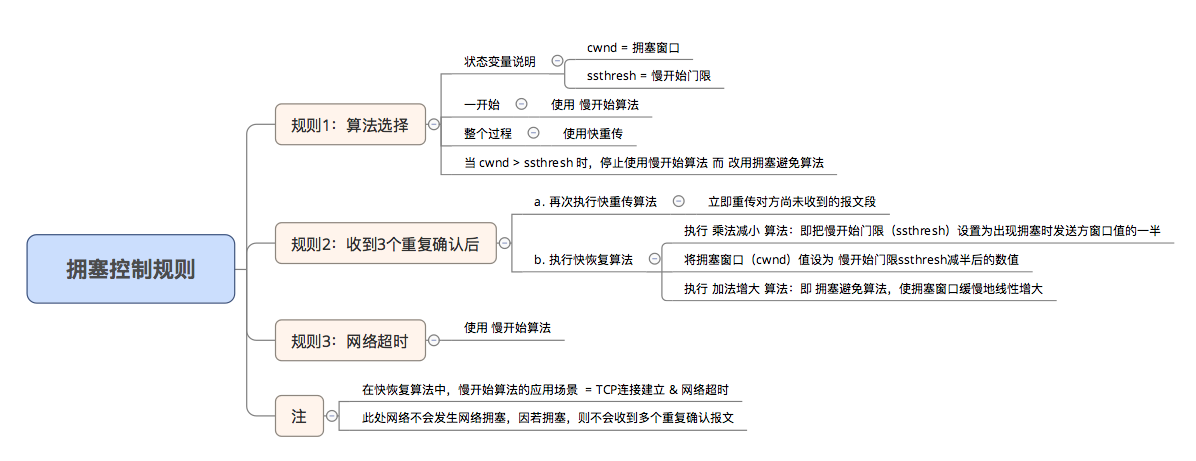
执行 **乘法减小** 算法：把 慢开始门限（ssthresh）设置为 出现拥塞时发送方窗口值的一半 = 拥塞窗口的1半

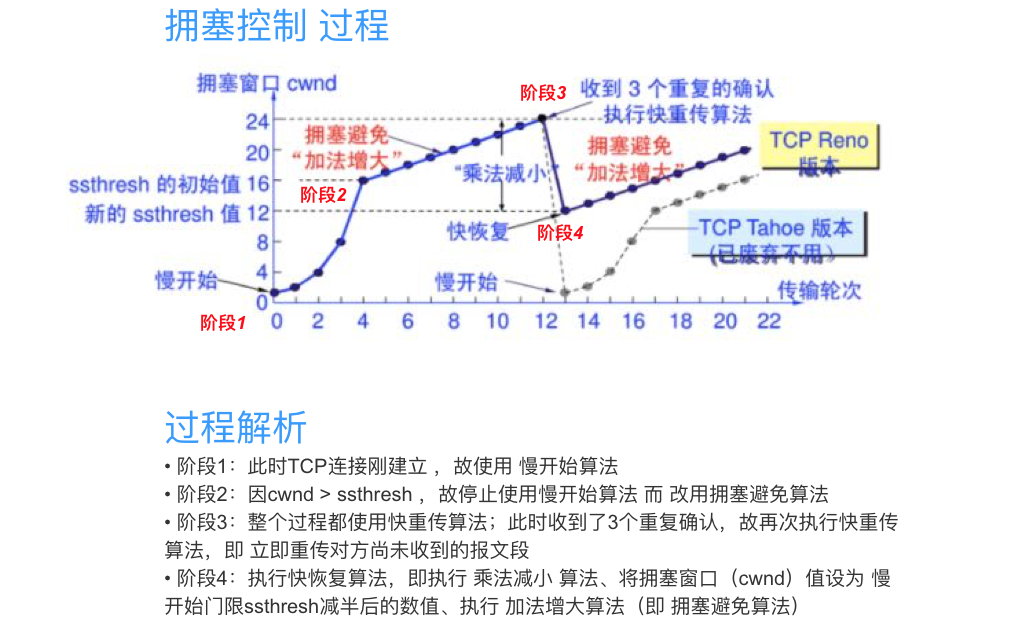
将拥塞窗口（cwnd）值设置为 慢开始门限ssthresh减半后的数值 = 拥塞窗口的1半

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

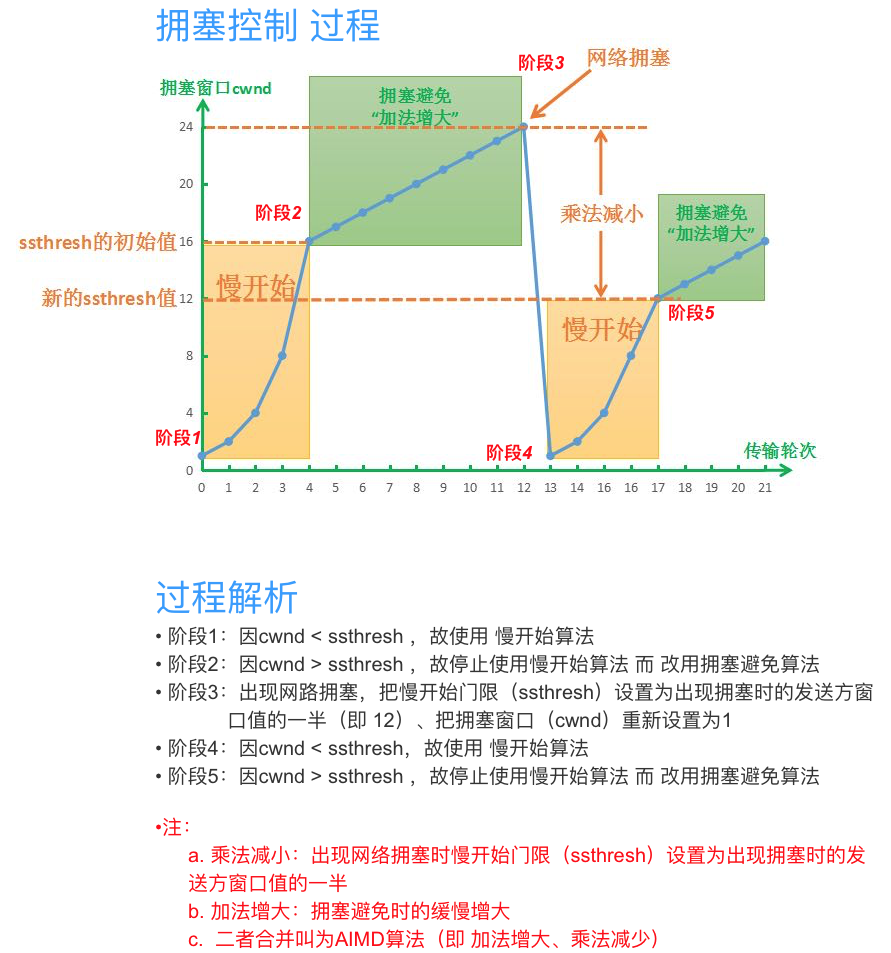
此处网络不会发生网络拥塞，因若拥塞，则不会收到多个重复确认报文





## 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）

## 加载一个class的过程

理解类在JVM中什么时候被加载和初始化是Java编程语言中的基础概念，正因为有了Java语言规范，我们才可以清晰的记录和解释这个问题，但是很多Java程序员仍然不知道什么时候类被加载，什么时候类被初始化，类加载和初始化好像让人很困惑，对初学者难以理解，在这篇教程中我们将看看类加载什么时候发生，类和接口是如何被初始化的，我并不会拘泥于类加载器的细节或者说类加载器的工作方式。仅仅使这篇文章更加专注和简结。

类什么时候加载

类的加载是通过类加载器（Classloader）完成的，它既可以是饿汉式[eagerly load]（只要有其它类引用了它就加载）加载类，也可以是懒加载[lazy load]（等到类初始化发生的时候才加载）。不过我相信这跟不同的JVM实现有关，然而他又是受JLS保证的（当有静态初始化需求的时候才被加载）。

#### 类什么时候初始化

加载完类后，类的初始化就会发生，意味着它会初始化所有类静态成员，以下情况一个类被初始化：

实例通过使用new()关键字创建或者使用class.forName()反射，但它有可能导致ClassNotFoundException。

类的静态方法被调用

类的静态域被赋值

静态域被访问，而且它不是常量

在顶层类中执行assert语句

反射同样可以使类初始化，比如java.lang.reflect包下面的某些方法，JLS严格的说明：一个类不会被任何除以上之外的原因初始化。

#### 类是如何被初始化的

现在我们知道什么时候触发类的初始化了，他精确地写在Java语言规范中。但了解清楚 域（fields，静态的还是非静态的）、块（block静态的还是非静态的）、不同类（子类和超类）和不同的接口（子接口，实现类和超接口）的初始化顺序也很重要类。事实上很多核心Java面试题和SCJP问题都是基于这些概念，下面是类初始化的一些规则：

类从顶至底的顺序初始化，所以声明在顶部的字段的早于底部的字段初始化

如果类的初始化是由于访问静态域而触发，那么只有声明静态域的类才被初始化，而不会触发超类的初始化或者子类的初始化即使静态域被子类或子接口或者它的实现类所引用。

接口初始化不会导致父接口的初始化。

静态域的初始化是在类的静态初始化期间，非静态域的初始化时在类的实例创建期间。这意味这静态域初始化在非静态域之前。

非静态域通过构造器初始化，子类在做任何初始化之前构造器会隐含地调用父类的构造器，他保证了非静态或实例变量（父类）初始化早于子类

#### 主动引用和被动引用

类的主动引用（一定会发生类的初始化）

new一个类的对象。

调用类的静态成员（除了final常最）和静态方法。

使用java.lang.reflect包的方法对类进行反射调用。

当虚拟机启动，java Hello则一定会初始化Hello类。说白了就是先启动main方法所在的类。

当初始化一个类，如果其父类没有被初始化，则先会初始化他的父类。

类的被动引用（不会发生类的初始化）

当访问一个静态域时，只有真正声明这个域的类才会被初始化。

通过子类引用父类的静态变量，不会导致子类初始化。

通过数组定义类引用，用不会触发此类的初始化。

引用常量不会触发此类的初始化（常量在编译阶段就存入调用类的常量池中了）

#### 初始化例子

这是一个有关类被初始化的例子，你可以看到哪个类被初始化

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | /\*\*   \* Java program to demonstrate class loading and initialization in Java.   \*/  public class ClassInitializationTest {        public static void main(String args[]) throws InterruptedException {            NotUsed o = null; //this class is not used, should not be initialized          Child t = new Child(); //initializing sub class, should trigger super class initialization          System.out.println((Object)o == (Object)t);      }  }    /\*\*   \* Super class to demonstrate that Super class is loaded and initialized before Subclass.   \*/  class Parent {      static { System.out.println("static block of Super class is initialized"); }      {System.out.println("non static blocks in super class is initialized");}  }    /\*\*   \* Java class which is not used in this program, consequently not loaded by JVM   \*/  class NotUsed {      static { System.out.println("NotUsed Class is initialized "); }  }    /\*\*   \* Sub class of Parent, demonstrate when exactly sub class loading and initialization occurs.   \*/  class Child extends Parent {      static { System.out.println("static block of Sub class is initialized in Java "); }      {System.out.println("non static blocks in sub class is initialized");}  }    Output:  static block of Super class is initialized  static block of Sub class is initialized in Java  non static blocks in super class is initialized  non static blocks in sub class is initialized  false |

从上面结果可以看出：

超类初始化早于子类

静态变量或代码块初始化早于非静态块和域

没使用的类根本不会被初始化，因为他没有被使用

再来看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | /\*\*   \* Another Java program example to demonstrate class initialization and loading in Java.   \*/    public class ClassInitializationTest {        public static void main(String args[]) throws InterruptedException {           //accessing static field of Parent through child, should only initialize Parent         System.out.println(Child.familyName);      }  }    class Parent {      //compile time constant, accessing this will not trigger class initialization      //protected static final String familyName = "Lawson";        protected static String familyName = "Lawson";        static { System.out.println("static block of Super class is initialized"); }      {System.out.println("non static blocks in super class is initialized");}  }    Output:  static block of Super class is initialized  Lawson |

分析：

这里的初始化发生是因为有静态域被访问，而且不一个编译时常量。如果声明的”familyName”是使用final关键字修饰的编译时常量使用（就是上面的注释代码块部分）超类的初始化就不会发生。

尽管静态与被子类所引用但是也仅仅是超类被初始化

还有另外一个例子与接口相关的，JLS清晰地解释子接口的初始化不会触发父接口的初始化。强烈推荐阅读JLS14.4理解类加载和初始化细节。以上所有就是有关类被初始化和加载的全部内容。

**1.加载：（重点）**  
加载阶段是“类加载机制”中的一个阶段，这个阶段通常也被称作“装载”，主要完成：  
1.通过“类全名”来获取定义此类的二进制字节流

2.将字节流所代表的静态存储结构转换为方法区的运行时数据结构

3.在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口

相对于类加载过程的其他阶段，加载阶段(准备地说，是加载阶段中获取类的二进制字节流的动作)是开发期可控性最强的阶段，因为加载阶段可以使用系统提供的类加载器(ClassLoader)来完成，也可以由用户自定义的类加载器完成，开发人员可以通过定义自己的类加载器去控制字节流的获取方式。

加载阶段完成后，虚拟机外部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，方法区中的数据存储格式有虚拟机实现自行定义，虚拟机并未规定此区域的具体数据结构。然后在java堆中实例化一个java.lang.Class类的对象，这个对象作为程序访问方法区中的这些类型数据的外部接口。

**2.验证：（了解）**

验证是链接阶段的第一步，这一步主要的目的是确保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安全。  
验证阶段主要包括四个检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。

1.文件格式验证

 验证class文件格式规范，例如： class文件是否已魔术0xCAFEBABE开头 ， 主、次版本号是否在当前虚拟机处理范围之内等

2.元数据验证

这个阶段是对字节码描述的信息进行语义分析，以保证起描述的信息符合java语言规范要求。验证点可能包括：这个类是否有父类(除了java.lang.Object之外，所有的类都应当有父类)、这个类是否继承了不允许被继承的类(被final修饰的)、如果这个类的父类是抽象类，是否实现了起父类或接口中要求实现的所有方法。

3.字节码验证

 进行数据流和控制流分析，这个阶段对类的方法体进行校验分析，这个阶段的任务是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：保证访法体中的类型转换有效，例如可以把一个子类对象赋值给父类数据类型，这是安全的，但不能把一个父类对象赋值给子类数据类型、保证跳转命令不会跳转到方法体以外的字节码命令上。

4.符号引用验证

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、符号引用类中的类，字段和方法的访问性(private、protected、public、default)是否可被当前类访问。

**3.准备：（了解）**

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的知识点，首先是这时候进行内存分配的仅包括类变量(static 修饰的变量),而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在java堆中。其次是这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量定义为:

public static int value  = 12;

那么变量value在准备阶段过后的初始值为0而不是12，因为这时候尚未开始执行任何java方法，而把value赋值为123的putstatic指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把value赋值为12的动作将在初始化阶段才会被执行。

上面所说的“通常情况”下初始值是零值，那相对于一些特殊的情况，如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值，建设上面类变量value定义为：

public static final int value = 123;

编译时javac将会为value生成ConstantValue属性，在准备阶段虚拟机就会根据ConstantValue的设置将value设置为123。

**4.解析：（了解）**  
解析阶段是虚拟机常量池内的符号引用替换为直接引用的过程。  
符号引用：符号引用是一组符号来描述所引用的目标对象，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标对象并不一定已经加载到内存中。

直接引用：直接引用可以是直接指向目标对象的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机内存布局实现相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同，如果有了直接引用，那引用的目标必定已经在内存中存在。

虚拟机规范并没有规定解析阶段发生的具体时间，只要求了在执行anewarry、checkcast、getfield、instanceof、invokeinterface、invokespecial、invokestatic、invokevirtual、multianewarray、new、putfield和putstatic这13个用于操作符号引用的字节码指令之前，先对它们使用的符号引用进行解析，所以虚拟机实现会根据需要来判断，到底是在类被加载器加载时就对常量池中的符号引用进行解析，还是等到一个符号引用将要被使用前才去解析它。

解析的动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。分别对应编译后常量池内的CONSTANT\_Class\_Info、CONSTANT\_Fieldref\_Info、CONSTANT\_Methodef\_Info、CONSTANT\_InterfaceMethoder\_Info四种常量类型。

1.类、接口的解析

2.字段解析

3.类方法解析

4.接口方法解析

**5.初始化：（了解）**

类的初始化阶段是类加载过程的最后一步，在准备阶段，类变量已赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器<clinit>()方法的过程。在以下四种情况下初始化过程会被触发执行：

1.遇到new、getstatic、putstatic或invokestatic这4条字节码指令时，如果类没有进行过初始化，则需先触发其初始化。生成这4条指令的最常见的java代码场景是：使用new关键字实例化对象、读取或设置一个类的静态字段(被final修饰、已在编译器把结果放入常量池的静态字段除外)的时候，以及调用类的静态方法的时候。

2.使用java.lang.reflect包的方法对类进行反射调用的时候

3.当初始化一个类的时候，如果发现其父类还没有进行过初始化、则需要先出发其父类的初始化

4.jvm启动时，用户指定一个执行的主类(包含main方法的那个类)，虚拟机会先初始化这个类

在上面准备阶段 public static int value  = 12;  在准备阶段完成后 value的值为0，而在初始化阶调用了类构造器<clinit>()方法，这个阶段完成后value的值为12。

\*类构造器<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static块)中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句快可以赋值，但是不能访问。

\*类构造器<clinit>()方法与类的构造函数(实例构造函数<init>()方法)不同，它不需要显式调用父类构造，虚拟机会保证在子类<clinit>()方法执行之前，父类的<clinit>()方法已经执行完毕。因此在虚拟机中的第一个执行的<clinit>()方法的类肯定是java.lang.Object。

\*由于父类的<clinit>()方法先执行，也就意味着父类中定义的静态语句快要优先于子类的变量赋值操作。

\*<clinit>()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句，也没有变量赋值的操作，那么编译器可以不为这个类生成<clinit>()方法。

\*接口中不能使用静态语句块，但接口与类不太能够的是，执行接口的<clinit>()方法不需要先执行父接口的<clinit>()方法。只有当父接口中定义的变量被使用时，父接口才会被初始化。另外，接口的实现类在初始化时也一样不会执行接口的<clinit>()方法。

\*虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步，如果多个线程同时去初始化一个类，那么只会有一个线程执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。如果一个类的<clinit>()方法中有耗时很长的操作，那就可能造成多个进程阻塞。

tack 是操作数栈）：