阿里面试集锦

一、红黑树的特性

（1）每个节点或者是黑色，或者是红色。
（2）根节点是黑色。
（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL 或 NULL)的叶子节点！]
（4）如果一个节点是红色的，则它的子节点必须是黑色的。
（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意：
(01) 特性(3)中的叶子节点，是只为空(NIL 或 null)的节点。
(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是 O(lgn)，效率非常之高。
例如，Java 集合中的 TreeSet 和 TreeMap，C++ STL 中的 set、map，以及 Linux 虚拟内存的管理，都是通过红黑树去实现的。

****

二、HashMap 和 HashTable 的不同点

1 继承和实现方式不同

HashMap 继承于 AbstractMap，实现了 Map、Cloneable、java.io.Serializable 接口。
Hashtable 继承于 Dictionary，实现了 Map、Cloneable、java.io.Serializable 接口。

2 线程安全不同

Hashtable 它是线程安全的，支持多线程。
而 HashMap 它不是线程安全的。

3 对 null 值的处理不同

HashMap 的 key、value 都可以为 **null**。
Hashtable 的 key、value 都不可以为 **null**。

4 支持的遍历种类不同

HashMap 只支持 **Iterator(迭代器)遍历。**
而 Hashtable 支持 **Iterator(迭代器)和 Enumeration(枚举器)两种方式遍历。**

5 通过 Iterator 迭代器遍历时，遍历的顺序不同

HashMap 是**"从前向后"**的遍历数组；再对数组具体某一项对应的链表，从表头开始进行遍历。
Hashtable 是**"从后往前"**的遍历数组；再对数组具体某一项对应的链表，从表头开始进行遍历。

6 容量的初始值 和 增加方式都不一样

HashMap **默认的容量大小是 16**；增加容量时，每次将容量变为**"原始容量 x2"**。
Hashtable **默认的容量大小是 11**；增加容量时，每次将容量变为"**原始容量 x2 + 1**"。

7 添加 key-value 时的 hash 值算法不同

HashMap 添加元素时，是使用**自定义的哈希算法。**
Hashtable 没有自定义哈希算法，而**直接采用的 key 的 hashCode()。**

8 部分 API 不同

Hashtable 支持 **contains(Object value)**方法，而且重写了 **toString()**方法；
而 HashMap 不支持 **contains(Object value)**方法，没有重写 **toString()**方法。

****

三、ConcurrentHashMap 为什么比 HashTable 性能好

ConcurrentHashMap 分段锁 Segment+HashEntry
HashTable 竞争同一个锁 Synchronized

Segment 类继承于 ReentrantLock，主要是为了使用 ReentrantLock 的锁，

ReentrantLock 的实现比 synchronized 在多个线程争用下的总体开销小

既然 ConcurrentHashMap 使用分段锁 Segment 来保护不同段的数据，那么在插入和获取元素的时候，必须先通过哈希算法定位到 Segment。可以看到 ConcurrentHashMap 会首先使用 Wang/Jenkins hash 的变种算法对元素的 hashCode 进行一次再哈希。
再哈希，其目的是为了减少哈希冲突，使元素能够均匀的分布在不同的 Segment 上，从而提高容器的存取效率。

整个操作是先定位到段，然后委托给段的 remove 操作。当多个删除操作并发进行时，只要它们所在的段不相同，它们就可以同时进行。

由于 put 方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须得加锁。Put 方法首先定位到 Segment，然后在 Segment 里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置然后放在 HashEntry 数组里。

- 是否需要扩容。在插入元素前会先判断 Segment 里的 HashEntry 数组是否超过容量（threshold），如果超过阀值，数组进行扩容。值得一提的是，Segment 的扩容判断比 HashMap 更恰当，因为 HashMap 是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时 HashMap 就进行了一次无效的扩容。

- 如何扩容。扩容的时候首先会创建一个两倍于原容量的数组，然后将原数组里的元素进行再 hash 后插入到新的数组里。为了高效 ConcurrentHashMap 不会对整个容器进行扩容，而只对某个 segment 进行扩容。

get 操作不需要锁。
除非读到的值是空的才会加锁重读，我们知道 HashTable 容器的 get 方法是需要加锁的，那么 ConcurrentHashMap 的 get 操作是如何做到不加锁的呢？原因是它的 get 方法里将要使用的共享变量都定义成 volatile。

size()操作

如果我们要统计整个 ConcurrentHashMap 里元素的大小，就必须统计所有 Segment 里元素的大小后求和。Segment 里的全局变量 count 是一个 volatile 变量，那么在多线程场景下，我们是不是直接把所有 Segment 的 count 相加就可以得到整个 ConcurrentHashMap 大小了呢？不是的，虽然相加时可以获取每个 Segment 的 count 的最新值，但是拿到之后可能累加前使用的 count 发生了变化，那么统计结果就不准了。所以最

安全的做法，是在统计 size 的时候把所有 Segment 的 put，remove 和 clean 方法全部锁住，但是这种做法显然非常低效。

因为在累加 count 操作过程中，之前累加过的 count 发生变化的几率非常小，所以 ConcurrentHashMap 的做法是先尝试 2 次通过不锁住 Segment 的方式来统计各个 Segment 大小，如果统计的过程中，容器的 count 发生了变化，则再采用加锁的方式来统计所有 Segment 的大小。

那么 ConcurrentHashMap 是如何判断在统计的时候容器是否发生了变化呢？使用 modCount 变量，在 put，remove 和 clean 方法里操作元素前都会将变量 modCount 进行加 1，那么在统计 size 前后比较 modCount 是否发生变化，从而得知容器的大小是否发生变化。

****

四、ClassLoader 的分类及加载顺序

主要分4类，见下图橙色部分

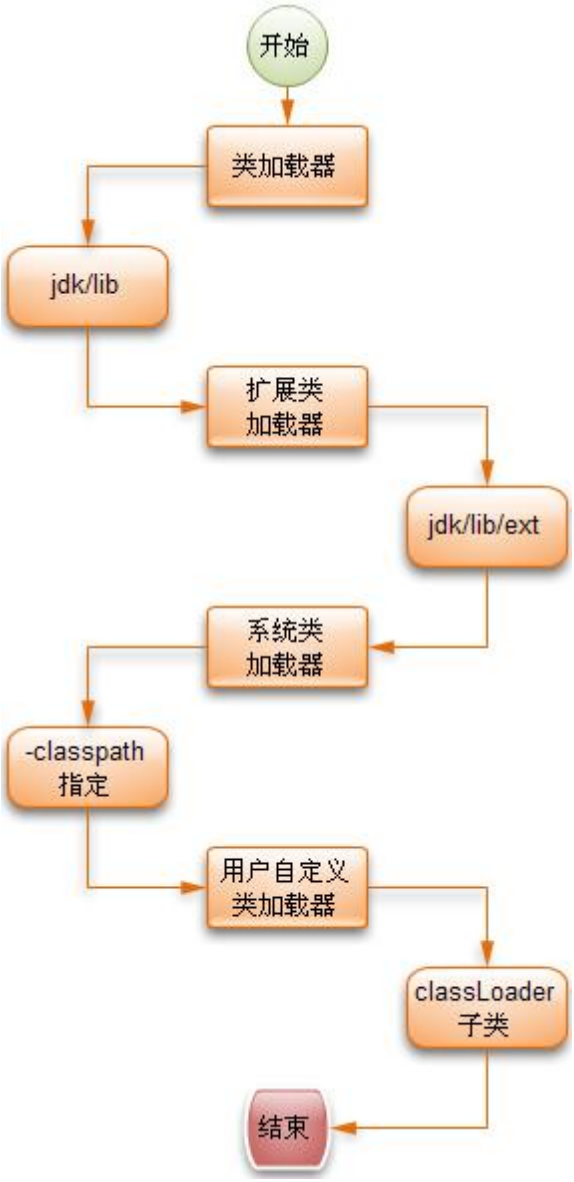
JVM类加载器：这个模式会加载JAVA_HOME/lib下的jar包

扩展类加载器：会加载JAVA_HOME/lib/ext下的jar包

系统类加载器：这个会去加载指定了classpath参数指定的jar文件

用户自定义类加载器：sun提供的ClassLoader是可以被继承的，允许用户自己实现类加载器

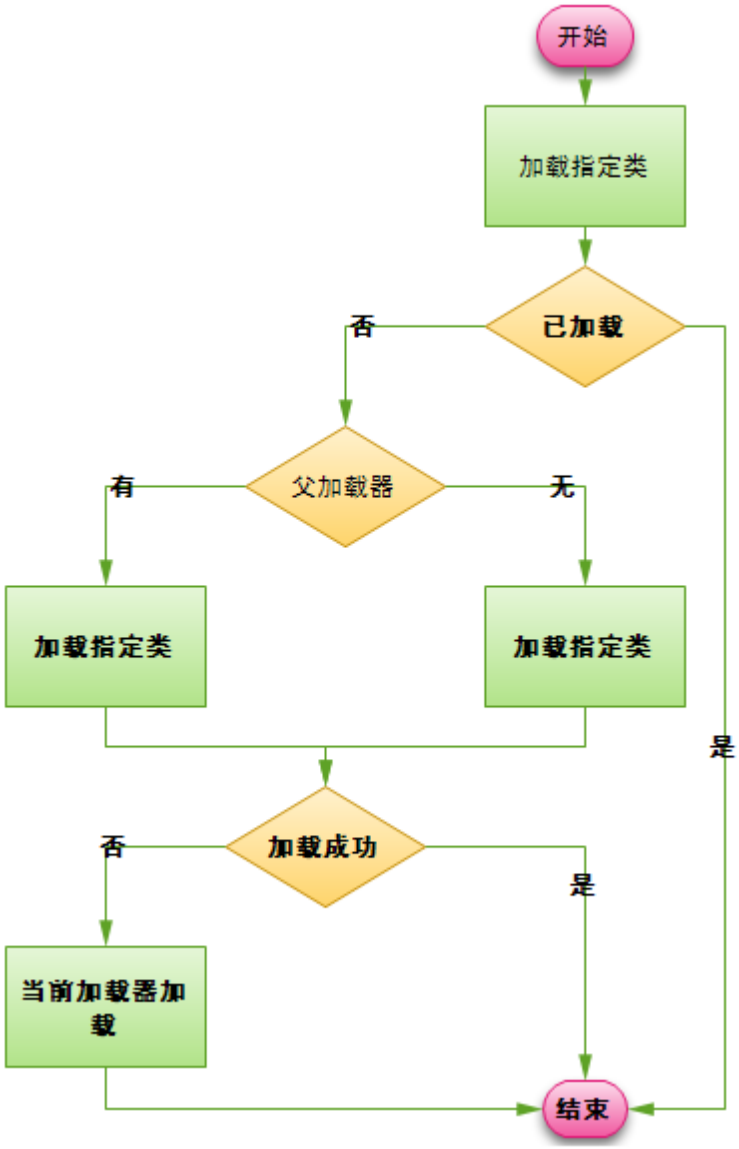
类加载器的加载顺序如图所示：



3.类加载顺序

JVM并不是把所有的类一次性全部加载到JVM中的，也不是每次用到一个类的时候都去查找，对于JVM级别的类加载器在启动时就会把默认的JAVA_HOME/lib里的class文件加载到JVM中，因为这些是系统常用的类，对于其他的第三方类，则采用用到时就去找，找到了就缓存起来的，下次再用到这个类的时候就可以直接用缓存起来的类对象了，ClassLoader之间也是有

父子关系的，没个ClassLoader都有一个父ClassLoader,在加载类时ClassLoader与其父ClassLoader的查找顺序如下图所示



****

五、数据库事物特性及隔离级别

如果一个数据库声称支持事务的操作，那么该数据库必须要具备以下四个特性：

⑴ 原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚，这和前面两篇博客介绍事务的功能是一样的概念，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

⑵ 一致性（Consistency）

一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态，也就是说一个事务执行之前和执行之后都必须处于一致性状态。

拿转账来说，假设用户 A 和用户 B 两者的钱加起来一共是 5000，那么不管 A 和 B 之间如何转账，转几次账，事务结束后两个用户的钱相加起来应该还得是 5000，这就是事务的一致性。

⑶ 隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

即要达到这么一种效果：对于任意两个并发的事务 T1 和 T2，在事务 T1 看来，T2 要么在 T1 开始之前就已经结束，要么在 T1 结束之后才开始，这样每个事务都感觉不到有其他事务在并发地执行。

关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。

⑷ 持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

例如我们在使用 JDBC 操作数据库时，在提交事务方法后，提示用户事务操作完成，当我们程序执行完成直到看到提示后，就可以认定事务以及正确提交，即使这时候数据库出现了问题，也必须要将我们的事务完全执行完成，否则就会造成我们看到提示事务处理完毕，但是数据库因为故障而没有执行事务的重大错误。

以上介绍完事务的四大特性(简称 ACID)，现在重点来说明下事务的隔离性，当多个线程都开启事务操作数据库中的数据时，数据库系统要能进行隔离操作，以保证各个线程获取数据的准确性，在介绍数据库提供的各种隔离级别之前，我们先看看如果不考虑事务的隔离性，会发生的几种问题：

1，脏读

脏读是指在一个事务处理过程里读取了另一个未提交的事务中的数据。

当一个事务正在多次修改某个数据，而在这个事务中这多次的修改都还未提交，这时一个并发的事务来访问该数据，就会造成两个事务得到的数据不一致。例如：用户 A 向用户 B 转账 100 元，对应 SQL 命令如下

```
update account set money=money+100 where name='B';  (此时 A 通知 B)

update account set money=money - 100 where name='A';
```

当只执行第一条 SQL 时，A 通知 B 查看账户，B 发现确实钱已到账（此时即发生了脏读），而之后无论第二条 SQL 是否执行，只要该事务不提交，则所有操作都将回滚，那么当 B 以后再次查看账户时就会发现钱其实并没有转。

2，不可重复读

不可重复读是指在对于数据库中的某个数据，一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了。

例如事务 T1 在读取某一数据，而事务 T2 立马修改了这个数据并且提交事务给数据库，事务 T1 再次读取该数据就得到了不同的结果，发送了不可重复读。

不可重复读和脏读的区别是，脏读是某一事务读取了另一个事务未提交的脏数据，而不可重复读则是读取了前一事务提交的数据。

在某些情况下，不可重复读并不是问题，比如我们多次查询某个数据当然以最后查询得到的结果为主。但在另一些情况下就有可能发生问题，例如对于同一个数据 A 和 B 依次查询就可能不同，A 和 B 就可能打起来了……

3，虚读(幻读)

幻读是事务非独立执行时发生的一种现象。例如事务 T1 对一个表中所有的行的某个数据项做了从"1"修改为"2"的操作，这时事务 T2 又对这个表中插入了一行数据项，而这个数据项的数值还是为"1"并且提交给数据库。而操作事务 T1 的用户如果再查看刚刚修改的数据，会发现还有一行没有修改，其实这行是从事务 T2 中添加的，就好像产生幻觉一样，这就是发生了幻读。

幻读和不可重复读都是读取了另一条已经提交的事务（这点就脏读不同），所不同的是不可重复读查询的都是同一个数据项，而幻读针对的是一批数据整体（比如数据的个数）。

现在来看看 MySQL 数据库为我们提供的四种隔离级别：

① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

③ Read committed (读已提交)：可避免脏读的发生。

④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

以上四种隔离级别最高的是 Serializable 级别，最低的是 Read uncommitted 级别，当然级别越高，执行效率就越低。像 Serializable 这样的级别，就是以锁表的方式(类似于 Java 多线程中的锁)使得其他的线程只能在锁外

等待，所以平时选用何种隔离级别应该根据实际情况。在 MySQL 数据库中默认的隔离级别为 Repeatable read (可重复读)。

在 MySQL 数据库中，支持上面四种隔离级别，默认的为 Repeatable read (可重复读)；而在 Oracle 数据库中，只支持 Serializable (串行化)级别和 Read committed (读已提交)这两种级别，其中默认的为 Read committed 级别。

六、常用的中间件：Redis 、zookeeper、MQ、dubbo 等

分布式锁实现

能说出一个熟悉的中间件原理

七、GC 算法、垃圾收集器

对象存活判断

判断对象是否存活一般有两种方式：

引用计数：每个对象有一个引用计数属性，新增一个引用时计数加 1，引用释放时计数减 1，计数为 0 时可以回收。此方法简单，**无法解决对象相互循环引用**的问题。

可达性分析（Reachability Analysis）：从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。不可达对象。

在 Java 语言中，GC Roots 包括：

虚拟机栈中引用的对象。

方法区中类静态属性实体引用的对象。

方法区中常量引用的对象。

本地方法栈中 JNI 引用的对象。

垃圾收集算法

标记 -清除算法

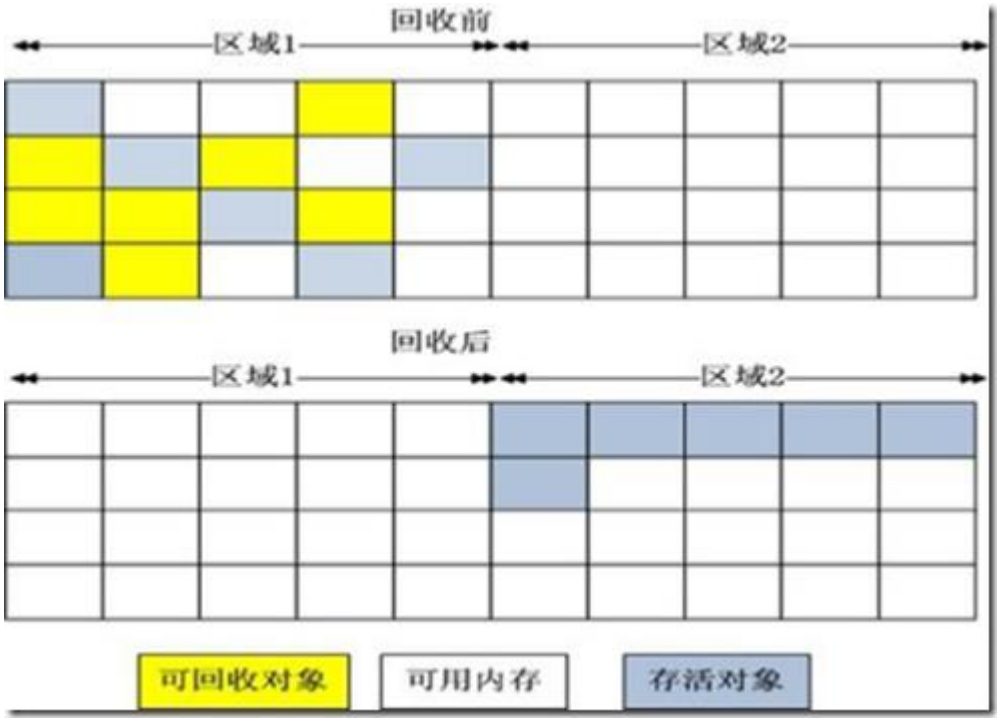
“**标记-清除**”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。

它的主要缺点有两个：一个是效率问题，标记和清除过程的效率都不高；另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，**空间碎片太多**可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

复制算法

“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

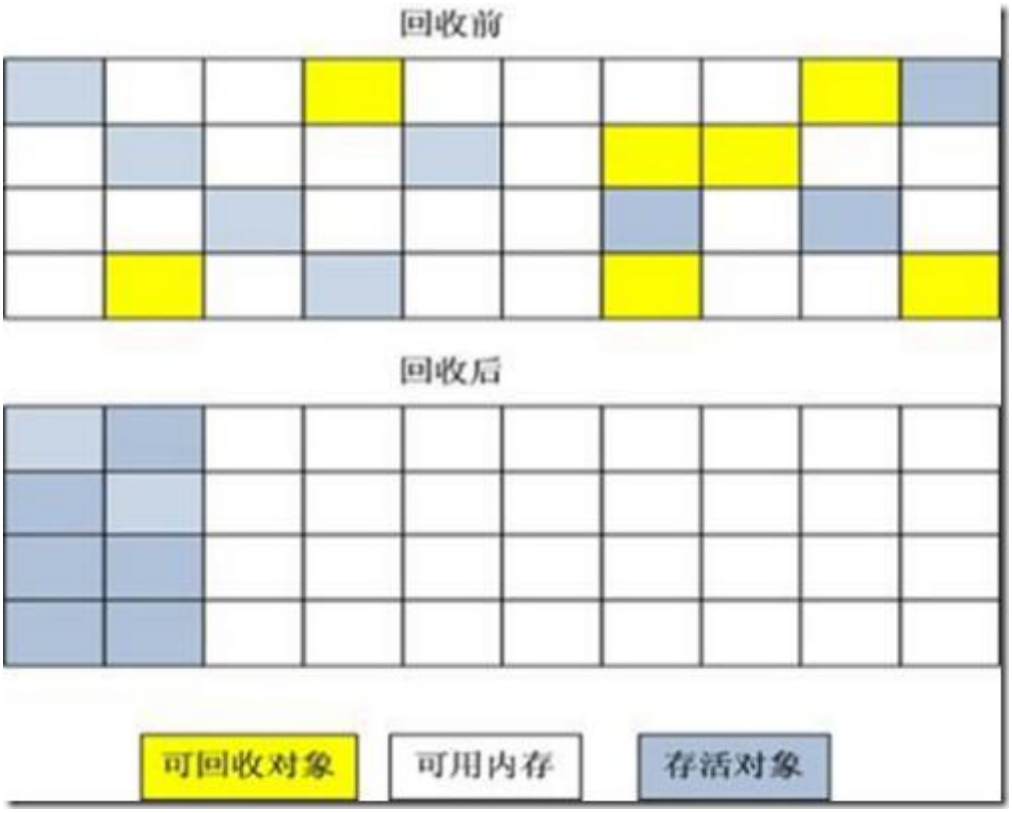
这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低。



标记-压缩算法

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存



分代收集算法

GC分代的基本假设：绝大部分对象的生命周期都非常短暂，存活时间短。

“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

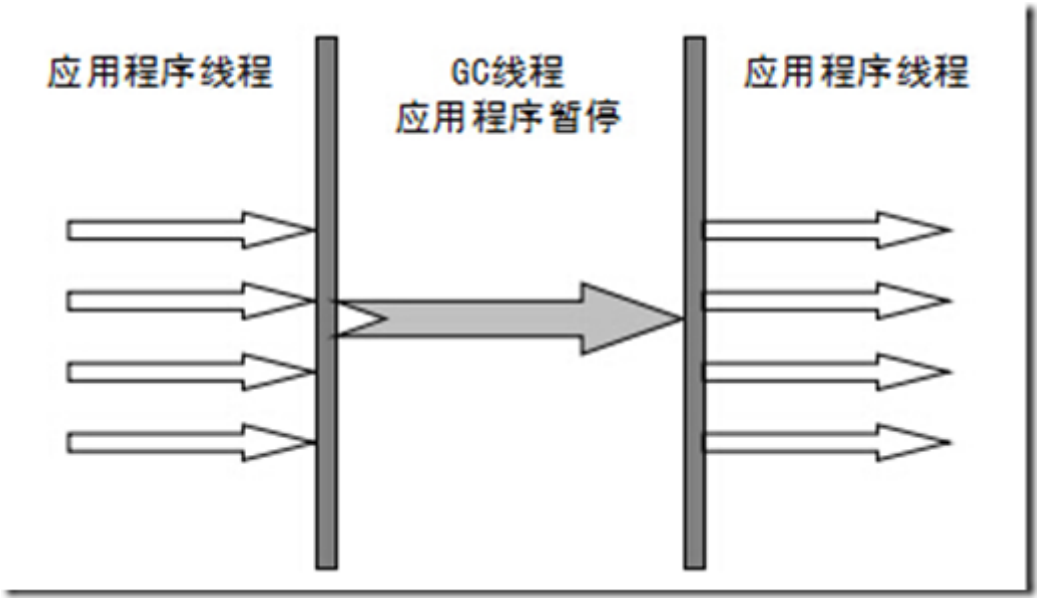
垃圾收集器

如果说收集算法是内存回收的方法论，垃圾收集器就是内存回收的具体实现

Serial 收集器

串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用**一个线程**去回收，新生代、老年代使用**串行**回收；**新生代复制算法**、**老年代标记-压缩**；垃圾收集的过程中会Stop The World（服务暂停）

参数控制：**-XX:+UseSerialGC** 串行收集器

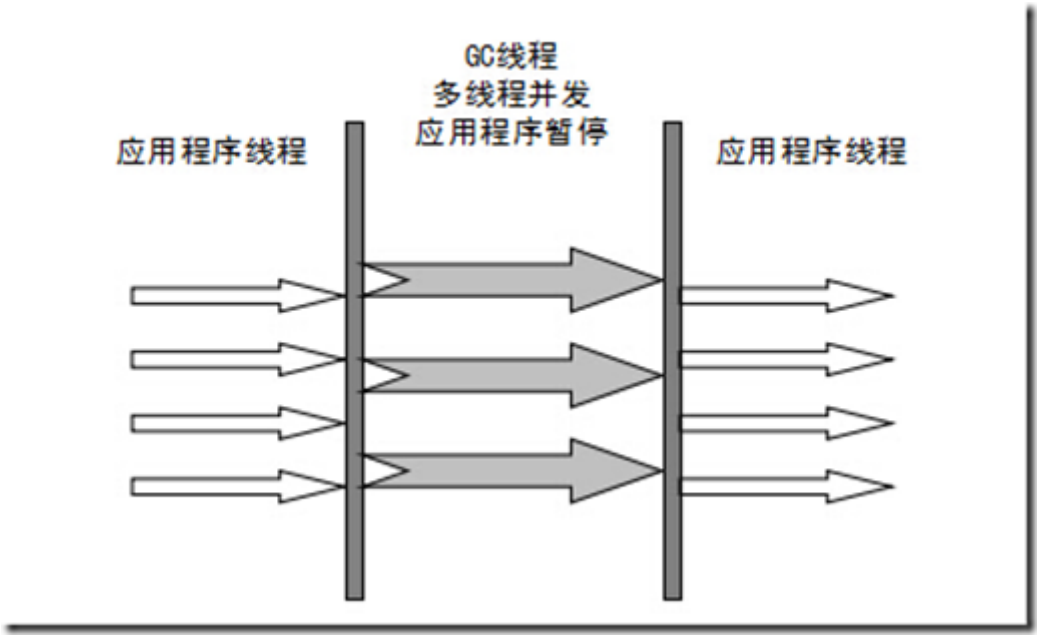


ParNew 收集器

ParNew收集器其实就是Serial收集器的多线程版本。**新生代并行，老年代串行；**新生代复制算法、老年代标记-压缩

参数控制：**-XX:+UseParNewGC** ParNew收集器

-XX:ParallelGCThreads 限制线程数量



Parallel 收集器

Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的**吞吐量**。可以通过参数来打开**自适应调节策略**，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以

提供最合适的停顿时间或最大的吞吐量；也可以通过参数控制GC的时间不大于多少毫秒或者比例；新生代复制算法、老年代标记-压缩

参数控制：**-XX:+UseParallelGC** 使用Parallel收集器+ 老年代串行

Parallel Old 收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记 - 整理”算法。这个收集器是在JDK 1.6中才开始提供

参数控制： **-XX:+UseParallelOldGC** 使用Parallel收集器+ 老年代并行

CMS 收集器

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用都集中在互联网站或B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

从名字（包含“Mark Sweep”）上就可以看出CMS收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为4个步骤，包括：

初始标记（CMS initial mark）

并发标记（CMS concurrent mark）

重新标记（CMS remark）

并发清除（CMS concurrent sweep）

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。　　由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行。**老年代收集器**（新生代使用ParNew）

优点:**并发收集**、**低停顿**

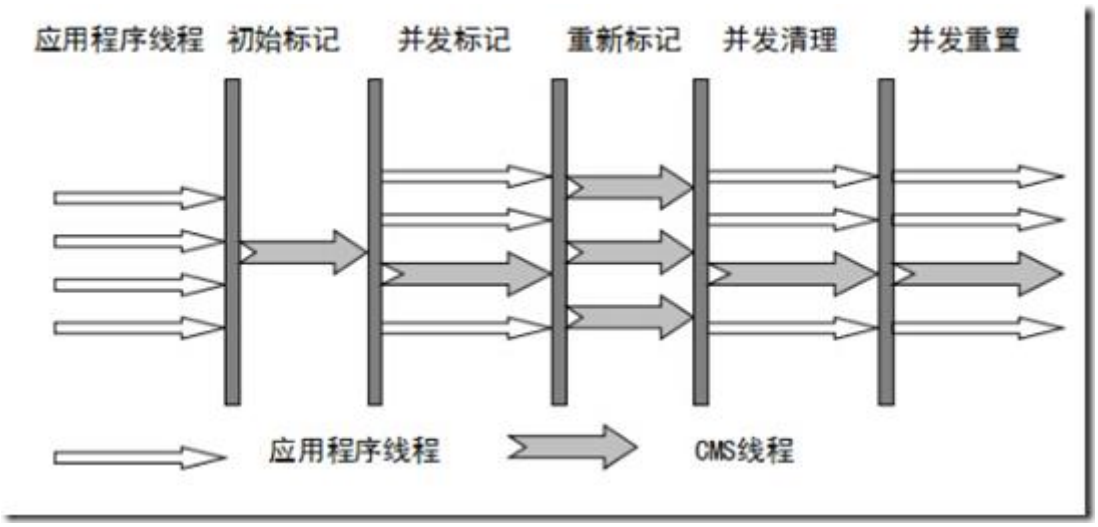
缺点：**产生大量空间碎片、并发阶段会降低吞吐量**

参数控制：**-XX:+UseConcMarkSweepGC** 使用CMS收集器

-XX:+ UseCMSCompactAtFullCollection Full GC后，进行一次碎片整理；整理过程是独占的，会引起停顿时间变长

-XX:+CMSFullGCsBeforeCompaction 设置进行几次Full GC后，进行一次碎片整理

-XX:ParallelCMSThreads 设定CMS的线程数量（一般情况约等于可用CPU数量）



****

CMS，全称Concurrent Low Pause Collector，是jdk1.4后期版本开始引入的新gc算法，在jdk5和jdk6中得到了进一步改进，它的主要适合场景是对响应时间的重要性需求 大于对吞吐量的要求，能够承受垃圾回收线程和应用线程共享处理器资源，并且应用中存在比较多的长生命周期的对象的应用。CMS是用于对tenured generation的回收，也就是年老代的回收，目标是尽量减少应用的暂停时间，减少full gc发生的几率，利用和应用程序线程并发的垃圾回收线程来标记清除年老代。在我们的应用中，因为有缓存的存在，并且对于响应时间也有比较高的要求，因此希 望能尝试使用CMS来替代默认的server型JVM使用的并行收集器，以便获得更短的垃圾回收的暂停时间，提高程序的响应性。

CMS收集周期

CMS并非没有暂停，而是用两次短暂停来替代串行标记整理算法的长暂停，它的收集周期是这样：

初始标记(CMS-initial-mark) -> 并发标记(CMS-concurrent-mark) -> 重新标记(CMS-remark) -> 并发清除(CMS-concurrent-sweep) ->并发重设状态等待下次CMS的触发(CMS-concurrent-reset)。

其中的1，3两个步骤需要暂停所有的应用程序线程的。第一次暂停从root对象开始标记存活的对象，这个阶段称为初始标记；第二次暂停是在并发标记之后， 暂停所有应用程序线程，重新标记并发标记阶段遗漏的对象（在并发标记阶段结束后对象状态的更新导致）。第一次暂停会比较短，第二次暂停通常会比较长，并且 remark这个阶段可以并行标记。

而并发标记、并发清除、并发重设阶段的所谓并发，是指一个或者多个垃圾回收线程和应用程序线程并发地运行，垃圾回收线程不会暂停应用程序的执行，如果你有多于一个处理器，那么并发收集线程将与应用线程在不同的处理器上运行，显然，这样的开销就是会降低应用的吞吐量。Remark阶段的并行，是指暂停了所有应用程序后，启动一定数目的垃圾回收进程进行并行标记，此时的应用线程是暂停的。

CMS的young generation的回收采用的仍然是并行复制收集器，这个跟Paralle gc算法是一致的。

参数介绍

1、启用CMS：-XX:+UseConcMarkSweepGC。

2。CMS默认启动的回收线程数目是（ParallelGCThreads + 3)/4），如果你需要明确设定，可以通过-XX:ParallelCMSThreads=20来设定，其中ParallelGCThreads是年轻代的并行收集线程数

3、CMS是不会整理堆碎片的，因此为了防止堆碎片引起full gc，通过会开启CMS阶段进行合并碎片选项：-XX:+UseCMSCompactAtFullCollection，开启这个选项一定程度上会影响性能，阿宝的blog里说也许可以通过配置适当的CMSFullGCsBeforeCompaction来调整性能，未实践。

4.为了减少第二次暂停的时间，开启并行remark: -XX:+CMSParallelRemarkEnabled。如果remark还是过长的话，可以开启-XX:+CMSScavengeBeforeRemark选项，强制remark之前开始一次minor gc，减少remark的暂停时间，但是在remark之后也将立即开始又一次minor gc。

5.为了避免Perm区满引起的full gc，建议开启CMS回收Perm区选项：+CMSPermGenSweepingEnabled -XX:+CMSClassUnloadingEnabled

6.默认CMS是在tenured generation沾满68%的时候开始进行CMS收集，如果你的年老代增长不是那么快，并且希望降低CMS次数的话，可以适当调高此值：

-XX:CMSInitiatingOccupancyFraction=80

这里修改成80%沾满的时候才开始CMS回收。

7.年轻代的并行收集线程数默认是(cpu <= 8) ? cpu : 3 + ((cpu * 5) / 8)，如果你希望降低这个线程数，可以通过-XX:ParallelGCThreads= N 来调整。

8.进入重点，在初步设置了一些参数后，例如：

```
-server -Xms1536m -Xmx1536m -XX:NewSize=256m -
XX:MaxNewSize=256m -XX:PermSize=64m
-XX:MaxPermSize=64m -XX:-UseConcMarkSweepGC -
XX:+UseCMSCompactAtFullCollection
-XX:CMSInitiatingOccupancyFraction=80 -
XX:+CMSParallelRemarkEnabled
-XX:SoftRefLRUPolicyMSPerMB=0
```

需要在生产环境或者压测环境中测量这些参数下系统的表现，这时候需要打开GC日志查看具体的信息，因此加上参数：

```
-verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -
Xloggc:/home/test/logs/gc.log
```

在运行相当长一段时间内查看CMS的表现情况，CMS的日志输出类似这样：

```
4391.322: [GC [1 CMS-initial-mark: 655374K(1310720K)]
662197K(1546688K), 0.0303050 secs] [Times: user=0.02
sys=0.02, real=0.03 secs]
```

```
4391.352: [CMS-concurrent-mark-start]
4391.779: [CMS-concurrent-mark: 0.427/0.427 secs] [Times:
user=1.24 sys=0.31, real=0.42 secs]
4391.779: [CMS-concurrent-preclean-start]
4391.821: [CMS-concurrent-preclean: 0.040/0.042 secs]
[Times: user=0.13 sys=0.03, real=0.05 secs]
4391.821: [CMS-concurrent-abortable-preclean-start]
4392.511: [CMS-concurrent-abortable-preclean: 0.349/0.690
secs] [Times: user=2.02 sys=0.51, real=0.69 secs]
4392.516: [GC[YG occupancy: 111001 K (235968 K)]4392.516:
[Rescan (parallel) , 0.0309960 secs]4392.547: [weak refs
processing, 0.0417710 secs] [1 CMS-remark:
655734K(1310720K)] 766736K(1546688K), 0.0932010 secs]
[Times: user=0.17 sys=0.00, real=0.09 secs]
4392.609: [CMS-concurrent-sweep-start]
4394.310: [CMS-concurrent-sweep: 1.595/1.701 secs]
[Times: user=4.78 sys=1.05, real=1.70 secs]
4394.310: [CMS-concurrent-reset-start]
4394.364: [CMS-concurrent-reset: 0.054/0.054 secs]
[Times: user=0.14 sys=0.06, real=0.06 secs]
```

其中可以看到CMS-initial-mark阶段暂停了0.0303050秒，而CMS-remark阶段暂停了0.0932010秒，因此两次暂停的总共时间是0.123506秒，也就是123毫秒左右。两次短暂停的时间之和在200以下可以称为正常现象。

但是你很可能遇到两种fail引起full gc：Prommotion failed和Concurrent mode failed。

Prommotion failed的日志输出大概是这样：

```
[ParNew (promotion failed): 320138K->320138K(353920K),
0.2365970 secs]42576.951: [CMS: 1139969K->1120688K(
```

```
166784K), 9.2214860 secs] 1458785K->1120688K(2520704K),
9.4584090 secs]
```

这个问题的产生是由于救助空间不够，从而向年老代转移对象，年老代没有足够的空间来容纳这些对象，导致一次full gc的产生。解决这个问题的办法有两种完全相反的倾向：增大救助空间、增大年老代或者去掉救助空间。 增大救助空间就是调整-XX:SurvivorRatio参数，这个参数是Eden区和Survivor区的大小比值，默认是32，也就是说Eden区是Survivor区的32倍大小，要注意Survivo是有两个区的，因此Surivivor其实占整个young genertation的1/34。调小这个参数将增大survivor区，让对象尽量在survitor区呆长一点，减少进入年老代的对象。去掉救助空 间的想法是让大部分不能马上回收的数据尽快进入年老代，加快年老代的回收频率，减少年老代暴涨的可能性，这个是通过将-XX:SurvivorRatio 设置成比较大的值（比如65536)来做到。在我们的应用中，将young generation设置成256M，这个值相对来说比较大了，而救助空间设置成默认大小(1/34)，从压测情况来看，没有出现prommotion failed的现象，年轻代比较大，从GC日志来看，minor gc的时间也在5-20毫秒内，还可以接受，因此暂不调整。

Concurrent mode failed 的产生是由于 CMS 回收年老代的速度太慢，导致年老代在 CMS 完成前就被沾满，引起 full gc，避免这个现象的产生就是调小-XX:CMSInitiatingOccupancyFraction 参数的值，让 CMS 更早更频繁的触发，降低年老代被沾满的可能。我们的应用暂时负

载比较低，在生产环境上年老代的增长非常缓慢，因此暂时设置此参数为 80。在压测环境下，这个参数的表现还可以，没有出现过 Concurrent mode failed。

****

常用的收集器组合

	新生代 GC 策略	年老代 GC 策略	说明
组合 1	Serial	Serial Old	Serial 和 Serial Old 都是单线程进行 GC，特点就是 GC 时暂停所有应用线程。
组合 2	Serial	CMS+Serial Old	CMS（Concurrent Mark Sweep）是并发 GC，实现 GC 线程和应用线程并发工作，不需要暂停所有应用线程。另外，当 CMS 进行 GC 失败时，会自动使用 Serial Old 策略进行 GC。
组合 3	ParNew	CMS	使用-XX:+UseParNewGC 选项来开启。ParNew 是 Serial 的并行版本，可以指定 GC 线程数，默认 GC 线程数为 CPU 的数量。可以使用-XX:ParallelGCThreads 选项指定 GC 的线程数。 如果指定了选项-XX:+UseConcMarkSweepGC 选项，则新生代默认使用 ParNew GC 策略。
组合 4	ParNew	Serial Old	使用-XX:+UseParNewGC 选项来开启。新生代使用 ParNew GC 策略，年老代默认使用 Serial Old GC 策略。
组合 5	Parallel Scavenge	Serial Old	Parallel Scavenge 策略主要是关注一个可控的吞吐量：应用程序运行时间 /（应用程序运行时间 + GC 时间），可见这会使得 CPU 的利用率尽可能的高，适用于后台持久运行的应用程序，而不适用于交互较多的应用程序。
组合 6	Parallel Scavenge	Parallel Old	Parallel Old 是 Serial Old 的并行版本
组合 7	G1GC	G1GC	-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC #开启 -XX:MaxGCPauseMillis =50 #暂停时间目标 -XX:GCPauseIntervalMillis =200 #暂停间隔

			目标 -XX:+G1YoungGenSize=512m #年轻代大小 - XX:SurvivorRatio=6 #幸存区比例

八、TCP

学习 TCP 协议，首先第一个要了解当然是 TCP 连接是如何建立的，下面给大家介绍一下三次握手和四次挥手的过程以及为什么要这样设计。

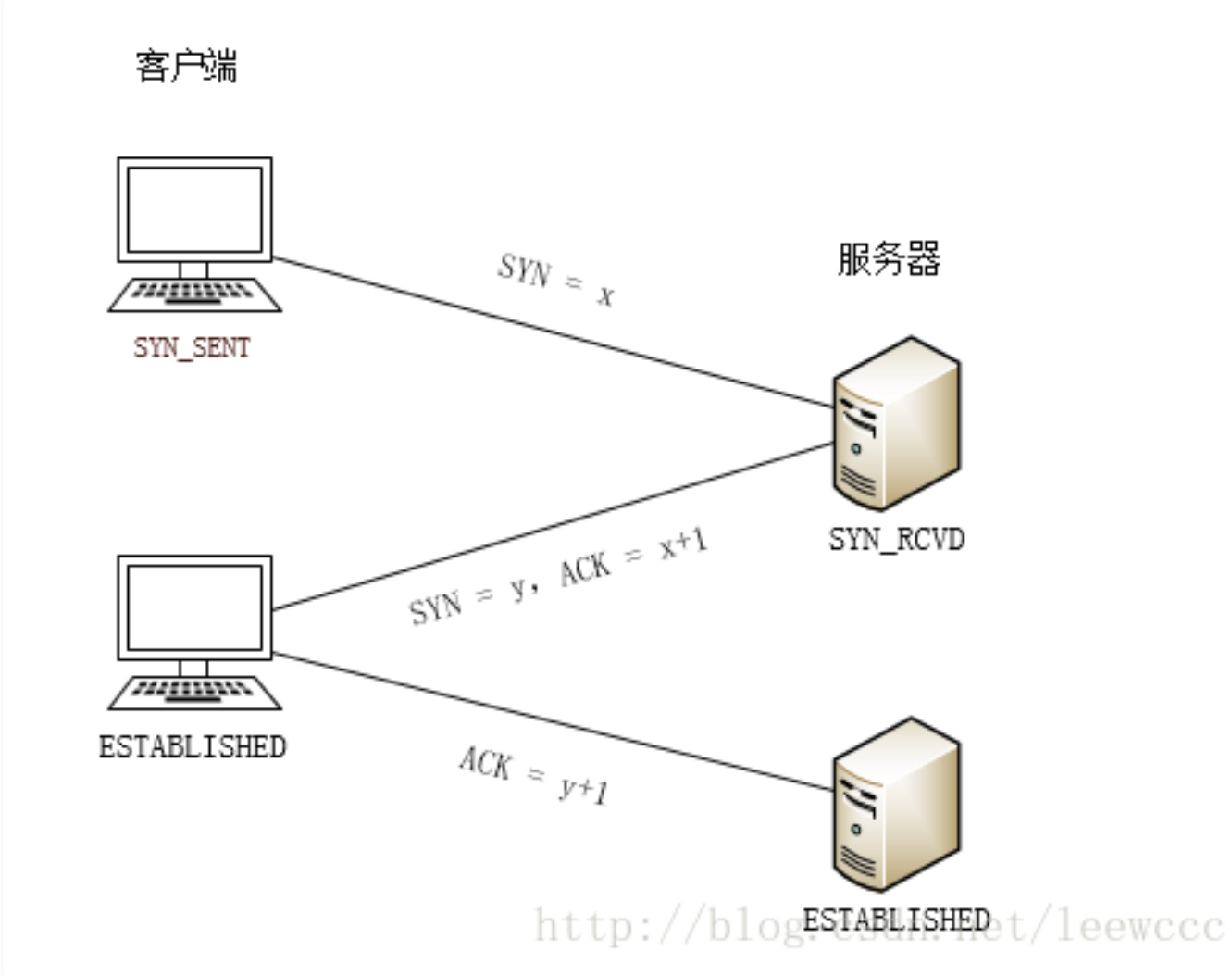
三次握手

在基于 TCP 通信中，双方要进行通信，则需要建立一个物理连接，建立时需要双方进行三次握手，成功即可完成连接建立。

采用三次握手的原因：

在网络通信中，网络存在拥塞，发送的报文可能会由于网络拥塞的原因，导致对方收不到。若采用直接开启连接，当客户端发送连接建立请求后，不等待确认服务器可以打开连接就直接打开连接，这样如果服务器收不到报文，根本不知道客户端，那么客户端的打开的物理连接是无效的，但客户端不知道，还一直发送数据，做无用的工作。

三次握手的过程



四次挥手

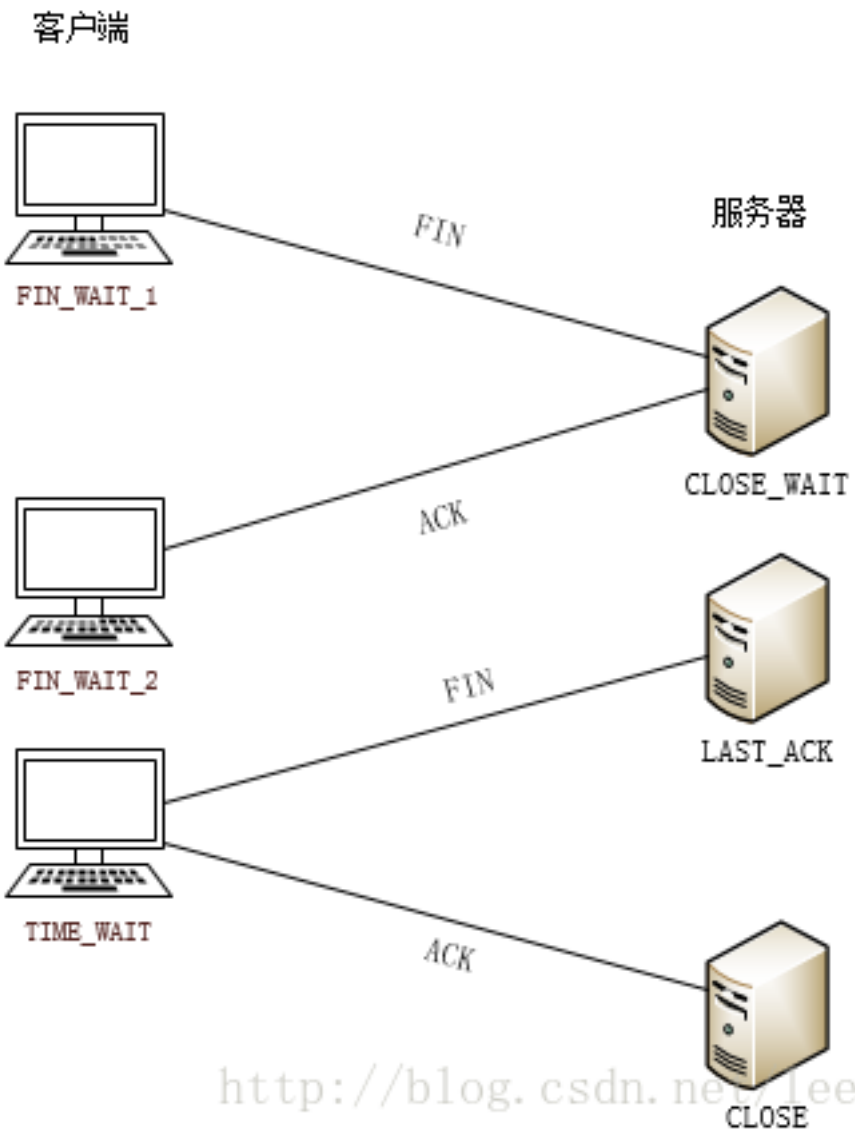
当双方通信结束时，需要四次挥手来关闭连接。

采用四次挥手的原因：

学习过 TCP 连接的都知道，TCP 连接是双向的，一个是从客户端到服务端，另一个是从服务端到客户端。假设当前客户端已经发送完所有数据到服务器，则此时可以告知服务器，我已经发送完数据了，可以关闭我这端到另一端的通道，服务器收到关闭报文则可发送一个确认，确认关闭；但此时由于服务器可能还需要发送数据到客户端，因此并不会关闭从服务端到客户端方向的通

道；等服务端发完了，才发送一个 FIN 报文给客户端，客户端收到之后发送确认，则此时 TCP 连接才正式关闭。

四次挥手的过程



TIME_WAIT 状态

从四次挥手过程可以看到，当服务器像客户端发送 FIN 报文后，客户端响应确认报文时，客户端处于 TIME_WAIT 状态，而不是处于 CLOSE 状态。之所以会这样主要是因为客户端发送确认报文后，不能立刻关闭连接。因为如果服务端收不到确认报文，会将 FIN 报文重传，但此时客户端已经关闭连接

了，这样会导致客户端收不到，而服务端则一直苦苦等待客户端发送确认报文，不断重传 FIN 报文。因此客户端在响应确认报文后，需要等待两个报文往返时间，以此来确保服务端能够正常收到确认报文关闭连接。

TCP的优势

从传输数据来讲，TCP/UDP以及其他协议都可以完成数据的传输，从一端传输到另外一端，TCP比较出众的一点就是提供一个**可靠的，流控的**数据传输，所以实现起来要比其他协议复杂的多，先来看下这两个修饰词的意义：

1. Reliability ，提供TCP的可靠性，TCP的传输要保证数据能够准确到达目的地，如果不能，需要能检测出来并且重新发送数据。
2. Data Flow Control，提供TCP的**流控**特性，管理发送数据的速率，不要超过设备的承载能力

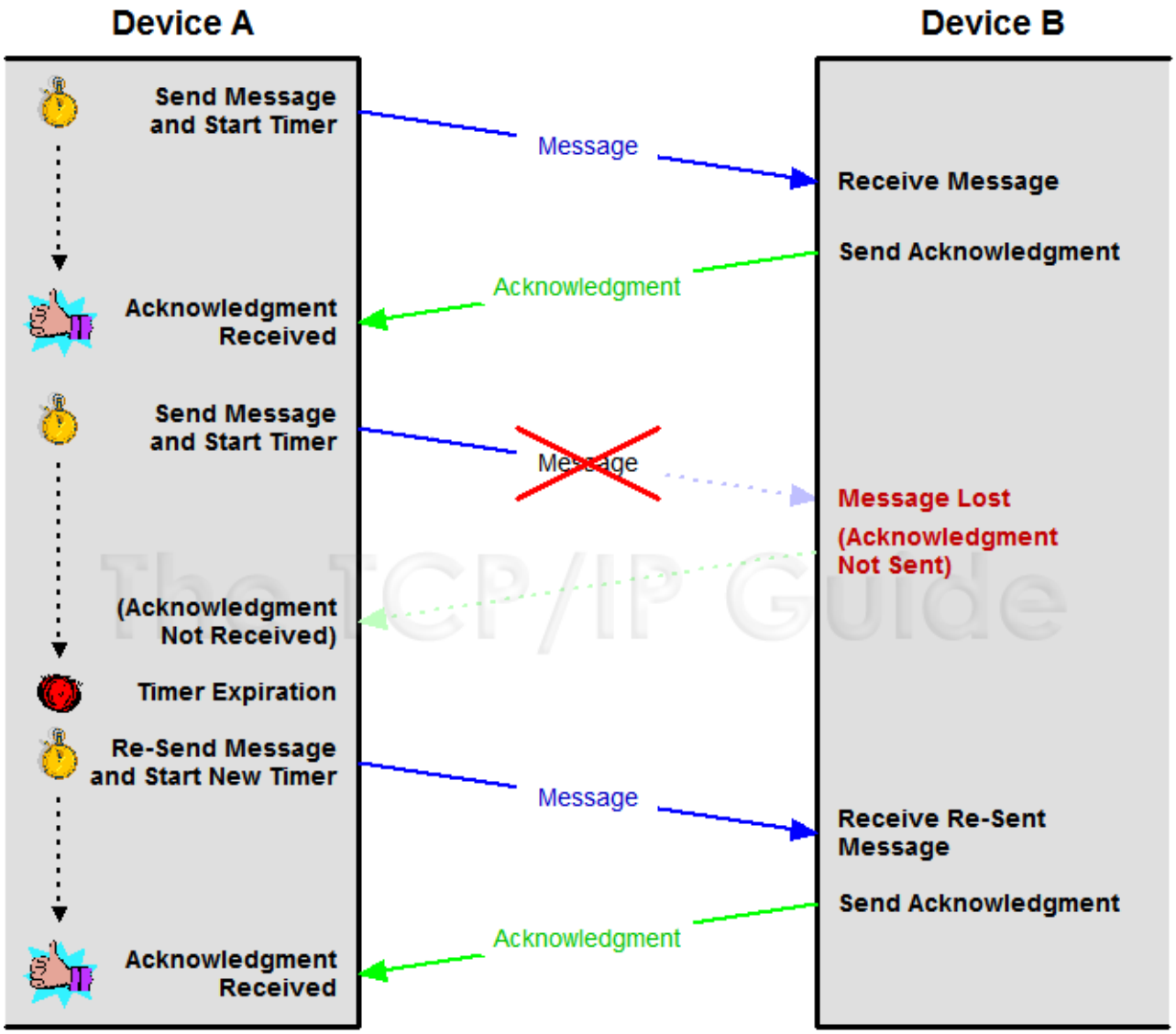
为了能够实现以上2点，TCP实现了很多细节的功能来保证数据传输，比如说 滑动窗口适应系统，超时重传机制，累计ACK等，这次先介绍一下滑动窗口的一些知识点。

滑动窗口引入

在阅读一些文章的时候看到一个大牛做的视频，非常不错易于理解滑动窗口的机制，可以先看下：

http://v.youku.com/v_show/id_XNDg1NDUyMDUy.html

IP层协议属于不可靠的协议，IP层并不关系数据是否发送到了对端，TCP通过确认机制来保证数据传输的可靠性，在比较早的时候使用的是**send--wait--send**的模式，其实这种模式叫做stop-wait模式，发送数据方在发送数据之后会启动定时器，但是如果数据或者ACK丢失，那么定时器到期之后，收不到ACK就认为发送出现状况，要进行重传。这样就会降低了通信的效率，如下图所示，这种方式被称为 positive acknowledgment with retransmission (PAR)



滑动窗口

可以假设一下，来优化一下PAR效率低的缺点，比如我让发送的每一个包都有一个id，接收端必须对每一个包进行确认，这样设备A一次多发送几个片段，而不必等候ACK，同时接收端也要告知它能够收多少，这样发送端发起来也有个限制，当然还需要保证顺序性，不要乱序，对于乱序的状况，我们可以允许等待一定情况下的乱序，比如说先缓存提前到的数据，然后去等待需要的数据，如果一定时间没来就DROP掉，来保证顺序性！

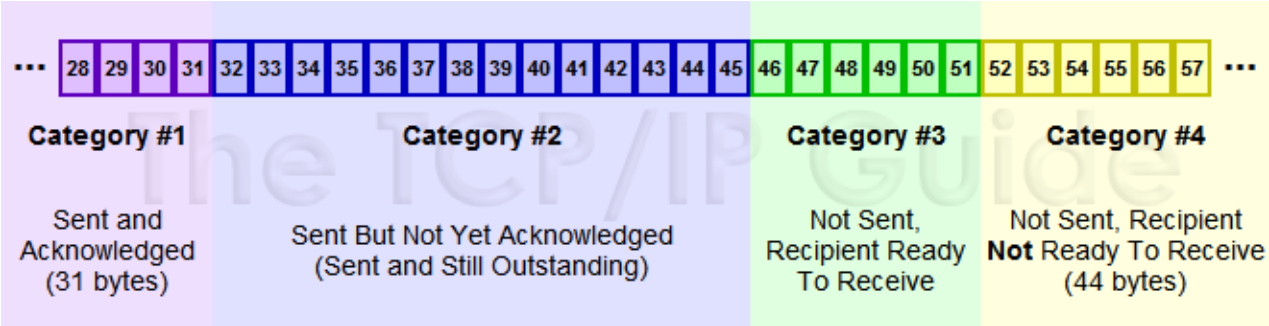
在TCP/IP协议栈中，滑动窗口的引入可以解决此问题，先来看从概念上数据分为哪些类

1. Sent and Acknowledged：这些数据表示已经发送成功并已经被确认的数据，比如图中的前31个bytes，这些数据其实的位置是在窗口之外了，因为窗口内顺序最低的被确认之后，要移除窗口，实际上是窗口进行合拢，同时打开接收新的带发送的数据

2. Send But Not Yet Acknowledged：这部分数据称为发送但没有被确认，数据被发送出去，没有收到接收端的ACK，认为并没有完成发送，这个属于窗口内的数据。

3. Not Sent，Recipient Ready to Receive：这部分是尽快发送的数据，这部分数据已经被加载到缓存中，也就是窗口中了，等待发送，其实这个窗口是完全有接收方告知的，接收方告知还是能够接受这些包，所以发送方需要尽快的发送这些包

4. Not Sent，Recipient Not Ready to Receive： 这些数据属于未发送，同时接收端也不允许发送的，因为这些数据已经超出了发送端所接收的范围



对于接收端也是有一个接收窗口的，类似发送端，接收端的数据有3个分类，因为接收端并不需要等待ACK所以它没有类似的接收并确认了的分类，情况如下

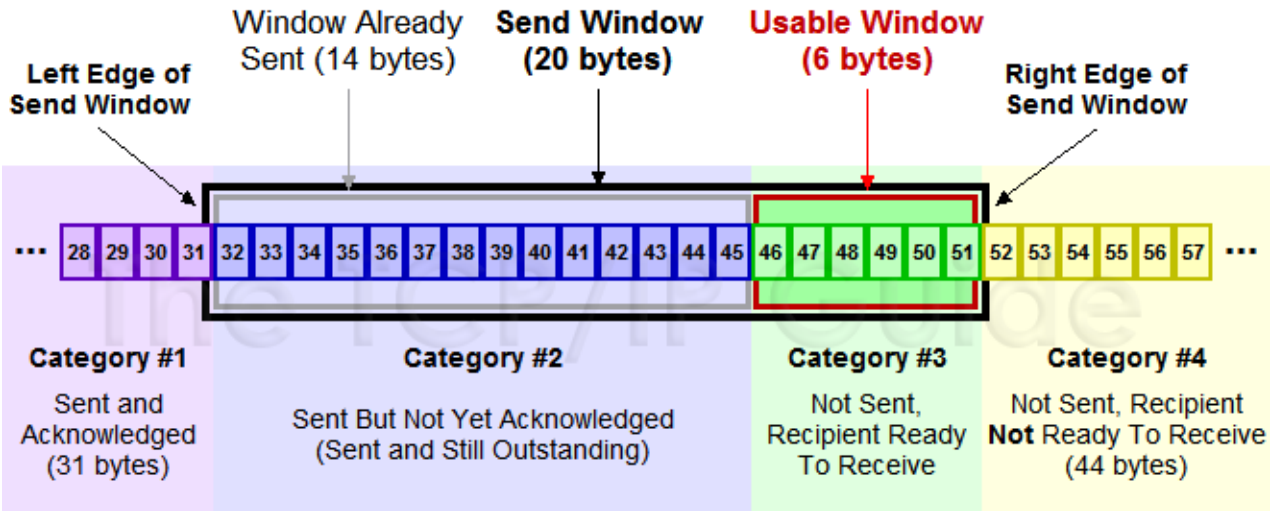
1. Received and ACK Not Send to Process：这部分数据属于接收了数据但是还没有被上层的应用程序接收，也是被缓存在窗口内
2. Received Not ACK: 已经接收并，但是还没有回复ACK，这些包可能输属于Delay ACK的范畴了
3. Not Received：有空位，还没有被接收的数据。

发送窗口和可用窗口

对于发送方来讲，窗口内的包括两部分，就是发送窗口（已经发送了，但是没有收到ACK），可用窗口，接收端允许发送但是没有发送的那部分称为可用窗口。

1. Send Window ： 20个bytes 这部分值是有接收方在三次握手的时候进行通告的，同时在接收过程中也不断的通告可以发送的窗口大小，来进行适应

2. Window Already Sent: 已经发送的数据，但是并没有收到ACK。



滑动窗口原理

TCP并不是每一个报文段都会回复ACK的，可能会对两个报文段发送一个ACK，也可能会对多个报文段发送1个ACK【累计ACK】，比如说发送方有1/2/3 3个报文段，先发送了2,3 两个报文段，但是接收方期望收到1报文段，这个时候2,3报文段就只能放在缓存中等待报文1的空洞被填上，如果报文1，一直不来，报文2/3也将被丢弃，如果报文1来了，那么会发送一个ACK对这3个报文进行一次确认。

举一个例子来说明一下滑动窗口的原理：

1. 假设32~45 这些数据，是上层Application发送给TCP的，TCP将其分成四个Segment来发往internet

2. seg1 32~34 seg3 35~36 seg3 37~41 seg4 42~45 这四个片段，依次发送出去，此时假设接收端之接收到了seg1 seg2 seg4

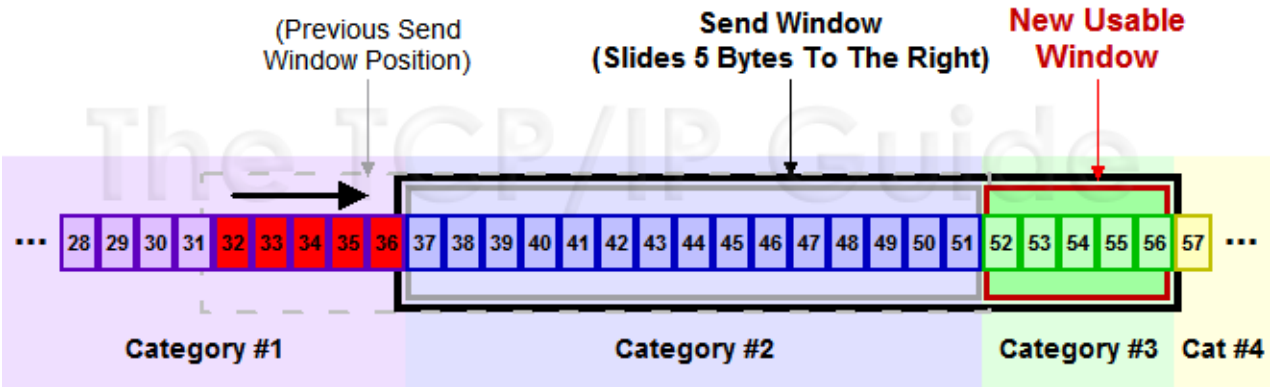
3. 此时接收端的行为是回复一个ACK包说明已经接收到了32~36的数据，并将seg4进行缓存（保证顺序，产生一个保存seg3 的hole）

4. 发送端收到ACK之后，就会将32~36的数据包从发送并没有确认切到发送已经确认，提出窗口，这个时候窗口向右移动

5. 假设接收端通告的Window Size仍然不变，此时窗口右移，产生一些新的空位，这些是接收端允许发送的范畴

6. 对于丢失的seg3，如果超过一定时间，TCP就会重新传送（重传机制），重传成功会seg3 seg4一块被确认，不成功，seg4也将被丢弃

就是不断重复着上述的过程，随着窗口不断滑动，将真个数据流发送到接收端，实际上接收端的Window Size通告也是会变化的，接收端根据这个值来确定何时及发送多少数据，从对数据流进行流控。原理图如下图所示：



滑动窗口动态调整

主要是根据接收端的接收情况，动态去调整Window Size，然后来控制发送端的数据流量

1. 客户端不断快速发送数据，服务器接收相对较慢，看下实验的结果

a. 包175，发送ACK携带WIN = 384，告知客户端，现在只能接收384个字节

b. 包176，客户端果真只发送了384个字节，Wireshark也比较智能，也宣告TCP Window Full

c. 包177，服务器回复一个ACK，并通告窗口为0，**说明接收方已经收到所有数据，并保存到缓冲区，但是这个时候应用程序并没有接收这些数据，导致缓冲区没有更多的空间**，故通告窗口为0, 这也就是所谓的**零窗口**，零窗口期间，发送方停止发送数据

d. 客户端察觉到窗口为0，则不再发送数据给接收方

e. 包178，接收方发送一个窗口通告，告知发送方已经有接收数据的能力了，可以发送数据包了

f. 包179，收到窗口通告之后，就发送缓冲区内的数据了.

No.	Time	Source	Destination	Protocol	Length	Info
171	0.086690	127.0.0.1	127.0.0.1	TCP	66	6666
172	0.086699	127.0.0.1	127.0.0.1	TCP	2962	57685
173	0.086747	127.0.0.1	127.0.0.1	TCP	66	6666
174	0.086750	127.0.0.1	127.0.0.1	TCP	1514	57685
175	0.126993	127.0.0.1	127.0.0.1	TCP	66	6666
176	0.334809	127.0.0.1	127.0.0.1	TCP	450	[TCP
177	0.334927	127.0.0.1	127.0.0.1	TCP	66	[TCP
178	0.421983	127.0.0.1	127.0.0.1	TCP	66	[TCP
179	0.421994	127.0.0.1	127.0.0.1	TCP	1514	57685
180	0.426743	127.0.0.1	127.0.0.1	TCP	66	6666
181	0.426755	127.0.0.1	127.0.0.1	TCP	2962	57685
182	0.426808	127.0.0.1	127.0.0.1	TCP	66	6666
183	0.426812	127.0.0.1	127.0.0.1	TCP	1514	57685
184	0.431033	127.0.0.1	127.0.0.1	TCP	66	6666

总结一点，就是接收端可以根据自己的状况通告窗口大小，从而控制发送端的接收，进行流量控制

九、**HTTPS**

HTTPS 要使客户端与服务器端的通信过程得到安全保证，必须使用的对称加密算法，但是协商对称加密算法的过程，需要使用非对称加密算法来保证安全，然而直接使用非对称加密的过程本身也不安全，会有中间人篡改公钥的可能性，所以客户端与服务器不直接使用公钥，而是使用数字证书签发机构颁发的证书来保证非对称加密过程本身的安全。这样通过这些机制协商出一个对称加密算法，就此双方使用该算法进行加密解密。从而解决了客户端与服务器端之间的通信安全问题。

十、CAP 原则

介绍

CAP 原则是 NOSQL 数据库的基石。Consistency（一致性）。 Availability（可用性）。Partition tolerance（分区容错性）[1]。

理论

分布式系统的 CAP 理论：理论首先把分布式系统中的三个特性进行了如下归纳：

- 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）
- 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）
- 分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 C 和 A 之间做出选择。

与可用的决择

CAP 理论就是说在分布式存储系统中，最多只能实现上面的两点。**而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容忍性是我们必须需要实现的。**所以我们只能在一致性和可用性之间进行权衡，没有 NoSQL 系统能同时保证这三点。

对于 web2.0 网站来说，关系数据库的很多主要特性却往往无用武之地

1. 数据库事务一致性需求

   很多 web 实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求并不高。允许实现最终一致性。

2. 数据库的写实时性和读实时性需求

   对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多 web 应用来说，并不要求这么高的实时性，比方说发一条消息之后，过几秒乃至十几秒之后，我的订阅者才看到这条动态是完全可以接受的。

3. 对复杂的 SQL 查询，特别是多表关联查询的需求

   任何大数据量的 web 系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的报表查询，特别是 SNS 类型的网站，从需求以及产品设计角 度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL 的功能被极大的弱化了。

与 NoSQL 的关系

传统的关系型数据库在功能支持上通常很宽泛，从简单的键值查询，到复杂的多表联合查询再到事务机制的支持。而与之不同的是，**NoSQL 系统通常注重性能和扩展性，而非事务机制**（事务就是强一致性的体现）[2]。

传统的 SQL 数据库的事务通常都是支持 ACID 的强事务机制。A 代表原子性，即在事务中执行多个操作是原子性的，要么事务中的操作全部执行，要么一个都不执行;C 代表一致性，即保证进行事务的过程中整个数据加的状态是一致的，不会出现数据花掉的情况;I 代表隔离性，即两个事务不会相互影响，覆盖彼此数据等;D 表示持久化，即事务一旦完成，那么数据应该是被写到安全的，持久化存储的设备上（比如磁盘）。

NoSQL 系统仅提供对行级别的原子性保证，也就是说同时对同一个 Key 下的数据进行的两个操作，在实际执行的时候是会串行的执行，保证了每一个 Key-Value 对不会被破坏。

与 BASE 的关系

BASE 就是为了解决关系数据库强一致性引起的问题而引起的可用性降低而提出的解决方案。

BASE 是下面三个术语的缩写：

- 基本可用（Basically Available）
- 软状态（Soft state）
- 最终一致（Eventually consistent）

1. 目前最快的 KV 数据库,10W 次/S，满足了高可用性。

2. Redis 的 k-v 上的 v 可以是普通的值（基本操作：get/set/del） v 可以是数值（除了基本操作之外还可以支持数值的计算） v 可以是数据结构比如基于链表存储的双向循环 list（除了基本操作之外还可以支持数值的计算，可以实现 list 的二头 pop,push）。如果 v 是 list，可以使用 redis 实现一个消息队列。如果 v 是 set,可以基于 redis 实现一个 tag 系统。与 mongodb 不同的地方是后者的 v 可以支持文档，比如按照 json 的结构存储。redis 也可以对存入的 Key-Value 设置 expire 时间。

3. Redis 的 v 的最大远远超过 memcache。这也是实现消息队列的一个前提。

十一、一致性哈希

一致性 hash 算法（DHT）通过减少影响范围的方式解决了增减服务器导致的数据散列问题，从而解决了分布式环境下负载均衡问题，如果存在热点数据，

那么通过增添节点的方式，对热点区间进行划分，将压力分配至其他服务器。重新达到负载均衡的状态。

欢迎关注我的微信公众号:"Java 面试通关手册"（一个有温度的微信公众号，期待与你共同进步~~~坚持原创，分享美文，分享各种 Java 学习资源）：。