的推移而快速增大。如果在 TCP 报文段丢失之后，发送方仍按照慢启动的方法发送 TCP 报文段，很有可能导致再次拥塞。

为了防止这一情况出现，**拥塞避免**（congestion avoidance）算法被提了出来。在进行重传时，该算法将慢启动阈值 *ssthresh*（slow start threshold）设置为 *cwnd* 值的一半，当 *cwnd* 增大到 *ssthresh* 之后，再减小每次收到 ACK 之后拥塞窗口的增大幅度。具体的公式如下所示。

$$cwnd = cwnd + MSS / cwnd$$

采用新的增大方法后，窗口大小会相对于 *RTT* 进行线性增大。最终的结果是，窗口大小会缓慢地增大到之前拥塞发生时的值。

此时的通信流程、滑动窗口和窗口大小的变化情况如图 3.10 所示。图中展示的，是假设当 *cwnd*=8 时发生了拥塞和重传，随后 *ssthresh* 被设为 4，然后经过慢启动，*cwnd* 到达 4 之后的整个流程示例[①]。

发送方首先发送 4 个 TCP 报文段（❶），随后在收到 4 个相应的 ACK 之后，将 *cwnd* 增大 1，并发送 5 个 TCP 报文段（❷）。之后，它以同样的思路，每次接收到个数与 *cwnd* 值相同的 ACK 之后，就将 *cwnd* 增大 1，慢慢地增加数据传输量（❸❹）。从窗口大小的变化就可以看出，慢启动阶段 *cwnd* 呈指数级增大，而在进入拥塞避免阶段（❶以后）后则呈线性增大，即增大趋势放缓了。换句话说，*cwnd* 增大到之前发生拥塞时候的值所用的时间更长了，所以可以传输更多的数据。

① 这里介绍的顺序其实是颠倒的，有关重传之后慢启动的流程请参考 3.5 节的内容。

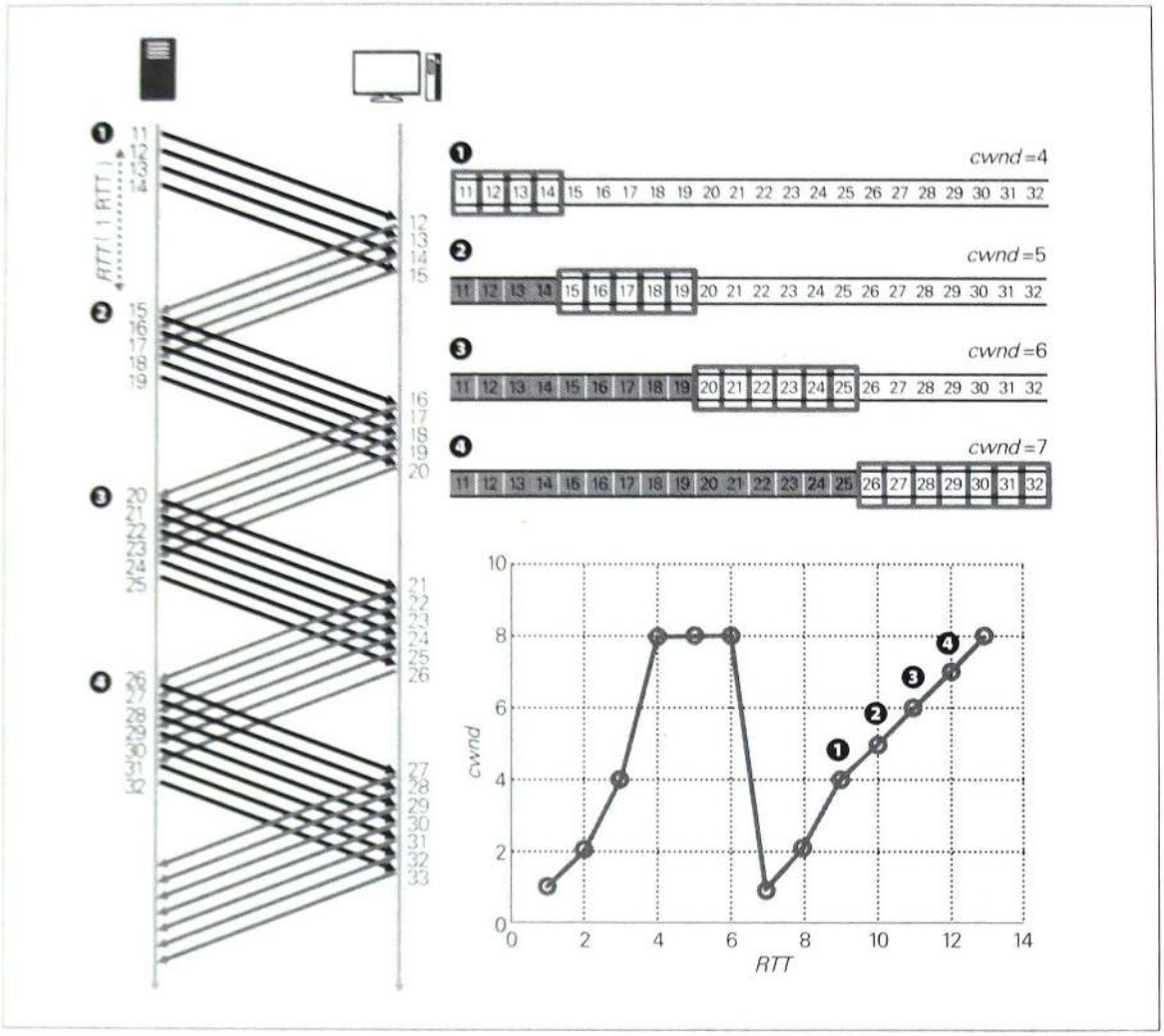

图3.10 拥塞避免

快速恢复

拥塞避免算法是为了使数据发送量不那么快到达会导致拥塞发生的大小而采取的一系列改良措施。此外，当TCP发现拥塞时，如果每次都重新从慢启动开始，效率肯定不高。因此，为了使重传开始时*cwnd*值不至于过小，一个用于提高传输效率的方案被提了出来。那便是**快速恢复**（fast recovery）。

人们通常认为快速恢复在拥塞并不严重的情况下比较有效。如果因为超时而发生重传，则考虑拥塞比较严重，所以使用慢启动更加合适。另外，如果发送方以收到**重复ACK**（duplicate ACK）为契机判断出现了轻度拥塞并进行重传控制，则是另外一种思路。这一思路称为快速重传（fast

retransmit），下一节将详细介绍它的具体流程。这里我们要知道的是，如果在拥塞并不严重的情况下使用慢启动开始重传，就会造成传输量下降的幅度过大，因此需要在进行窗口控制时使用快速恢复算法，以保持一定程度的数据传输量。

使用快速恢复算法时拥塞窗口的变化情况如图 3.11 所示。由于该算法的具体流程需要和快速重传一起详细介绍，所以请大家稍后参考 3.6 节以实际的拥塞控制算法 Reno、NewReno 为例进行的说明。

如图 3.11 所示，假设当时间（t）为 6 时进行了快速重传，那么发送方需要将窗口大小 *cwnd* 减半，同时将此值作为 *ssthresh* 保存下来，然后将 *cwnd* 增大 3 个 TCP 报文段大小。之所以增大 3 个 TCP 报文段大小，是因为在快速重传时，重传过程是以收到 3 个重复的 ACK 为契机开始的，因此发送方可以认为至少有 3 个 TCP 报文段已经发送到了接收方（详见图 3.14）。随后，在重传的 TCP 报文段送达接收方之前，发送方会一直收到重复的 ACK。但是，每次收到重复 ACK 之后，只增加 1 个单位的窗口大小，同时如果窗口内有尚未发送的 TCP 报文段，就发送出去。因此，从 $t=6$ 开始，*cwnd* 会暂时小幅增大，以避免在检测到拥塞之后出现吞吐量过度降低的现象。在 $t=8$ 时，发送方收到了与重传的 TCP 报文段对应的新 ACK，于是将 *cwnd* 设置为 *ssthresh* 的值，并进入拥塞避免阶段。

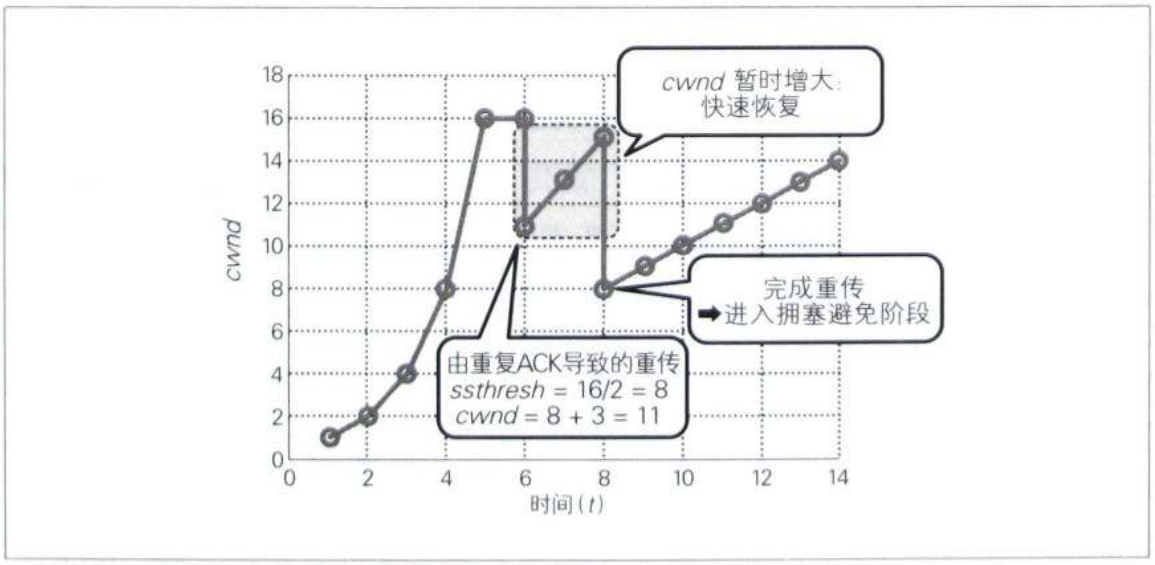

图 3.11 快速恢复

TCP拥塞控制算法如前文所述，进行下列两种处理。

- 提升传输量，使其尽可能接近拥塞的临界值
- 在检测到拥塞之后重新开始传输数据时，不过度降低传输量

从以上可以看出，TCP拥塞控制算法的目的就是更高效率地传输数据。Reno便使用了这个快速恢复算法。此外，NewReno算法还针对“快速重传～快速恢复”阶段发生的多个TCP报文段丢失的问题，进行了相应的优化。

3.5 重传控制

高可靠性传输的关键——准确且高效

要想将数据完整无误地发送给对方，**重传**是最为重要的手段。此外，要想实现高效率的数据传输，TCP必须干净利落，尽早检查出丢包情况，快速地进行处理。

本节将从思路和功能两方面介绍如何实现准确且高效的重传控制。

高可靠性传输所需的重传控制

如果网络中的TCP报文段或者ACK丢失，那么对应的TCP报文段就会被重传。可以毫无疑问地说，重传是确保传输可靠性的最重要的功能。但同时，传输效率也必须纳入考虑范围之内。以上这些特性是如何作为通信协议的一部分搭载在通信设备中的呢？

当出现以下两种情形时，便可以认为TCP报文段丢失了。

- 重传计时器超时
- 收到多个重复的ACK

接下来我们试着以上述两种情形为契机，重传丢失的TCP报文段。

下文将介绍“❶基于重传计时器的超时控制”和“❷使用重复 ACK”两种重传控制算法。

❶基于重传计时器的超时控制

检测数据是否丢失的一个方法是，“判断 ACK 是否到达了发送方”。使用计时器就可以完成检测与判断。在发送完 TCP 报文段之后，发送方便设置一个与该报文段对应的计时器，当发现一段时间内仍没有收到对应 ACK，则代表出现了超时的情况，发送方就会重新发送 TCP 报文段。

实际的流程与滑动窗口的变化情况如图 3.12 所示。当拥塞窗口大小 *cwnd* 为 8 时，发送方为各个 TCP 报文段设置对应的计时器（❶）。

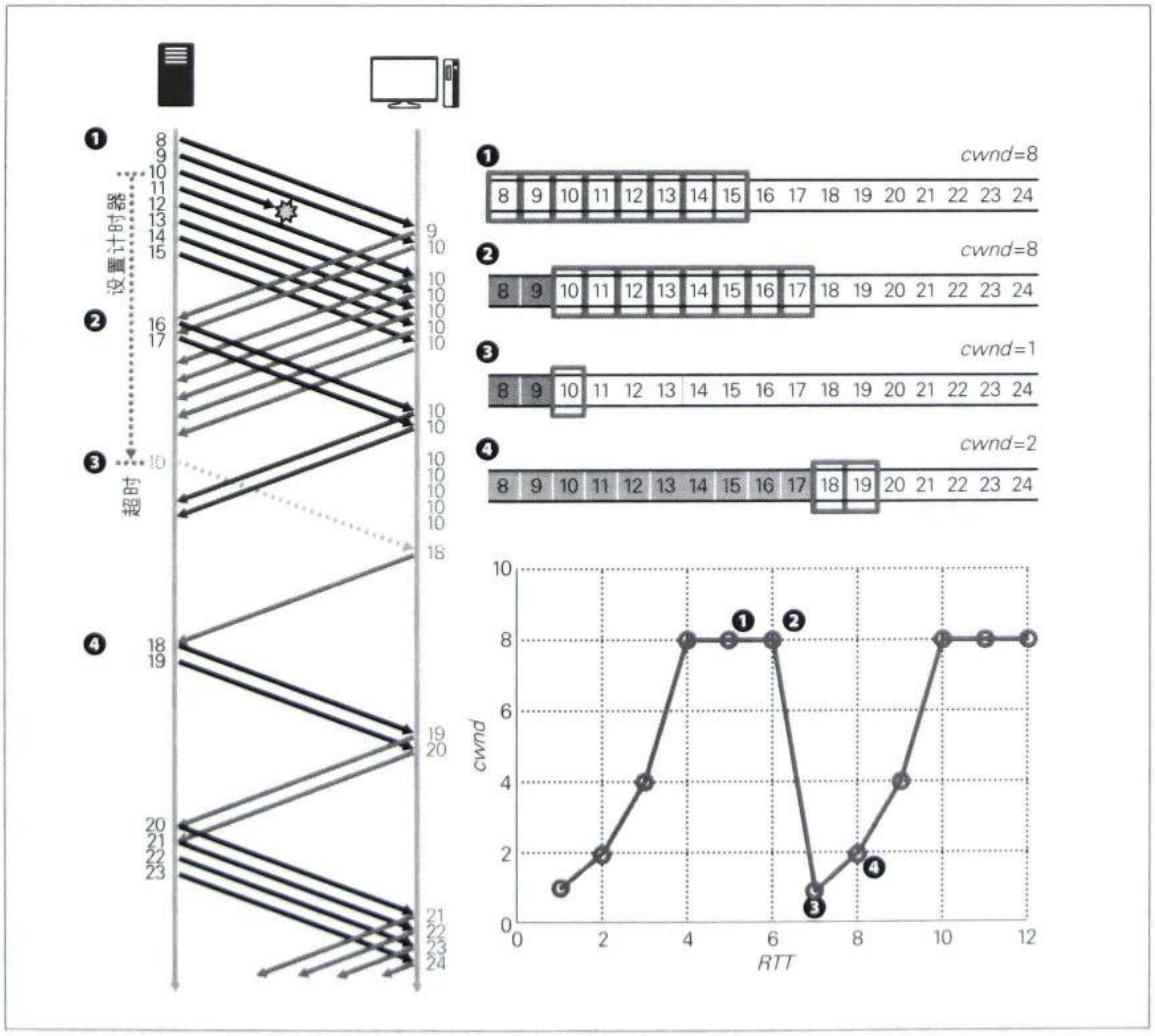

图 3.12 超时后的重传

这里，我们假设10号TCP报文段丢失了。由于发送方已经收到了8号和9号TCP报文段对应的ACK，也就是说8号和9号TCP报文段都已经成功发送，所以就将窗口往右滑动，继续发送尚未发送的TCP报文段（❷）。将窗口大小8设为最大值，此时就不再增大窗口大小。

随后，接收方持续发送请求10号报文段的ACK，而发送方由于在此情况下无法滑动窗口，所以无法发送之后的TCP报文段。在此期间，10号TCP报文段对应的重传计时器超时，因此发送方重传10号TCP报文段，同时将*cwnd*设置为初始值1（❸）。

之后从慢启动重新开始传输数据。当重传的TCP报文段顺利到达接收方，接收方会返回请求18号TCP报文段的ACK（也就是说，告诉发送方自己已经收到了17号之前的TCP报文段）。接下来，按照慢启动的窗口控制算法，发送方每次收到ACK之后让*cwnd*增大1个单位大小（❹）。

虽然每次发送TCP报文段都会设置对应的重传计时器，但确定合适的***RTO***（超时重传时间）非常重要。如果该值过大，会导致传输效率过低；反之，如果该值过小，就会导致虽然可以正确地传输，但是会频繁进行不必要的重传，最终给网络带来额外的压力。从数据发送出去开始到收到ACK为止的往返时间*RTT*非常容易受到网络拥堵情况、链路长短等因素的影响。

因此，随时根据*RTT*的值计算出*RTO*的值，才可以动态地适应*RTT*的变化。而实现这一点所必需的，便是根据快速变化的*RTT*计算出*RTO*的方法。下文将介绍两种计算*RTO*值的算法。

一——以往基于SRTT的求值方法 RTO计算方法(1)

TCP标准文档RFC 743中规定了根据*SRTT*（Smoothed Round Trip Time，平滑往返时延）求*RTO*的方法，其中*SRTT*是由*RTT*的测量值平滑计算得到的。此方法的目的是动态地适应*RTT*值的变化情况。*SRTT*和*RTO*可以通过以下公式进行计算。

$$SRTT = \alpha \cdot SRTT + (1-\alpha) \cdot RTT$$
$$RTO = \beta \cdot SRTT$$

其中，α 称为平滑因子，RFC 793 中的推荐值是 0.8～0.9；β 称为时延变化因子，推荐值是 1.3～2.0。整个过程就是，首先计算 *RTT* 的值，然后更新 *SRTT* 的值，最后根据 *SRTT* 决定到底要使用多大的 *RTO* 值。

——Jacobson 提出的新算法 RTO 计算方法②

然而，上述的超时重传时间计算方法有个问题，那就是没有考虑 *RTT* 值的偏差。如果 *RTT* 的偏差值过大，那么由前面的公式计算出来并进行平滑化后得到的值是无法反映这种剧烈变化的。其结果就是有时 *RTO* 比 *RTT* 还要小。也就是说，即使发送方已经成功发送了 TCP 报文段，也会因为错误的判断产生不必要的重传。

因此，1988 年范·雅各布森（Van Jacobson）提出了新的超时重传时间计算方法，这种方法将 *RTT* 的方差纳入了计算，基于下列公式计算 *RTO* 的值。

$$
\begin{aligned}
Err &= RTT - SRTT \\
SRTT &= SRTT + g1 \cdot Err \\
v &= v + g2 \cdot (|Err| - v) \\
RTO &= SRTT + 4 \cdot v
\end{aligned}
$$

SRTT 使用 *Err* 进行平滑化处理。*Err* 代表 *SRTT* 与 *RTT* 的误差，接下来它会被用作平均评估值。*SRTT* 的计算是通过每次将 *g*1 乘以误差值加权平均计算得到的。而 *v* 是平均偏差值，是通过每次将 *g*2 乘以误差与平均偏差的差值并加权平均计算后得到的。最后，通过联合 *SRTT* 与平均偏差值进行计算，可得到 *RTO*。

之所以计算平均偏差而非标准方差，是因为计算标准方差需要计算平方根，而如果通过平均偏差去计算 *RTO*，能大幅度减少计算量。此外，*g*1 和 *g*2 两个平滑因子都有对应的推荐值，*g*1 是 0.125，*g*2 则是 2.25。至于为何计算 *RTO* 时要在 *SRTT* 上加上 4 倍的平均偏差值，是因为这里假定所有的 *RTT* 值都收敛在平均偏差值 4 倍以内。

此方法考虑到了 *RTT* 的方差，因此也适合 *RTT* 大幅变化的情况，现

在实际运行的算法中大部分采用了此技术。

——随 RTT 变化而产生的不同表现

图 3.13 展示的是当 *RTT* 发生变化时，前面两种 *RTO* 计算方法表现出来的不同结果。以往的 *RTO* 计算方法 [图中以 *RTO*（Original）表示] 无法应对 *RTT* 发生剧烈变化的情形，从图中可以看到会出现 *RTO* < *RTT* 的情况。换句话说，这意味着将会有不必要的重传出现。显而易见，考虑了方差的 Jacobson 的算法 [图中以 RTO（Jacobson）表示] 可以有效地捕捉到 *RTT* 急速增大的变化，能够规避无意义的重传。

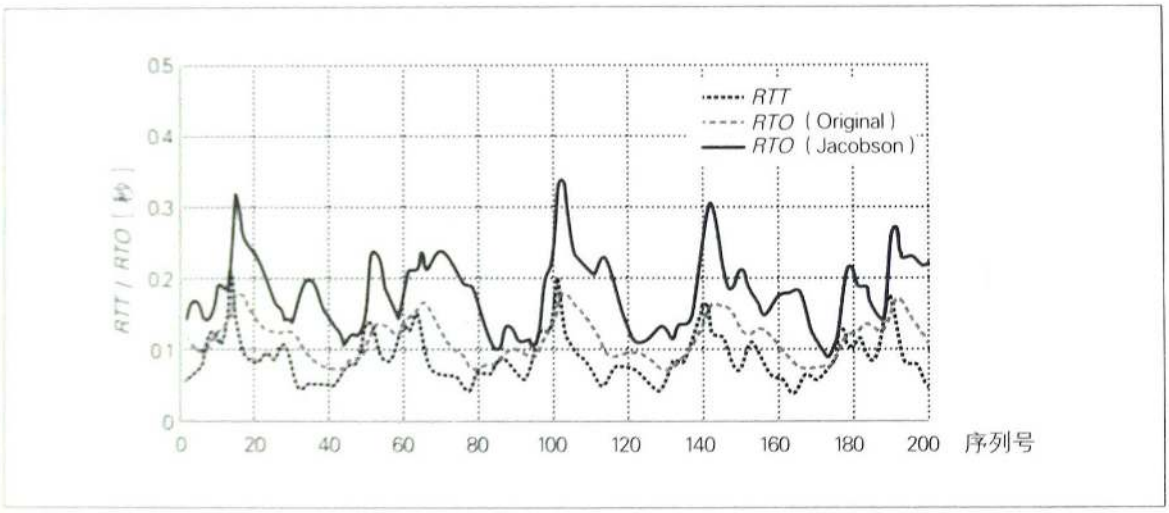

图 3.13 RTT 和 RTO 的变化情况

TCP 标准会在收到 ACK 之后计算 *RTT*，但是这里有一个问题。由于 ACK 返回需要一段时间，所以如果进行了无意义的重传，那么很有可能会出现这样的情况：使用了重传的 TCP 报文段和与之前发送的报文段对应的 ACK 来计算 *RTT* 值。这会导致 *RTT* 值极小，最终使得 *RTO* 的值也不正确。因此对于重传的 TCP 报文段，就不再计算 *RTT* 值，以防止这种误操作出现。然而，在这种情况下 *RTO* 就无法更新，反而会导致 *RTO* 的值无法有效地反映 *RTT* 的变化情况。

因此这里引入了一项安全策略，那就是在发生重传时，将 *RTO* 的值变为 2 倍。如果连续发生重传，*RTO* 就会指数级增大，这样下去会导致重传的等待时间很长，进而导致效率降低。*RTO* 最多只能增加到 64 秒，如

果超过这个时间，重传仍然失败，协议会认为网络或者接收方主机出现了异常，并强制断开连接。

2 使用重复 ACK 快速重传算法

要完成快速恢复，前提就是保证**快速重传算法**的运行。然而，快速恢复与拥塞避免有关，而快速重传与重传控制有关，两者是相互独立的功能。Tahoe 中只搭载了快速重传，其下一个版本 Reno 中则搭载了快速重传和快速恢复。3.6 节将详细介绍两者的运行机制。

假如总是以超时来判断 TCP 报文段是否丢失，有时需要等待太长的时间，这会导致效率降低。因此研究者提出了使用 ACK 的高效率重传控制方法。当 TCP 报文段丢失之后，接收方接收到的是与期望接收的序列号不同的数据。在这种情况下，接收方会一直发送请求未接收序列号数据的 ACK，直到收到为止，而发送方则一直重复收到请求同一个序列号的 ACK。快速重传算法便是利用了这一特点。

发送方如果连续 3 次收到与此前相同的 ACK，就认为当前 TCP 报文段已经丢失，可以无须等待计时器超时就重传数据。相比超时重传机制，此算法速度更快，因此被称为快速重传。RFC 2581 中记载有此算法和后文介绍的快速恢复算法的详细内容。之所以使用数字 3，主要是因为第 1 次和第 2 次收到重复 ACK 也可能只是因为 TCP 报文段发生了乱序。这只是为了尽可能地减少不必要的重传。

快速重传的逻辑流程如图 3.14 所示。这里，我们假设在发送 8 个 TCP 报文段时，其中的 12 号报文段丢失了。发送方在收到请求 12 号报文段的 ACK 之前，虽然可以发送新的报文段，但是在发送之后会因为无法滑动窗口而导致数据传输停止。当发送方重复收到 3 次请求 12 号报文段的 ACK 时（也就是说，总共收到 4 个 ACK），发送方会认为 12 号报文段已经丢失，并进行重传处理。采用此方法，发送方便可以无须等待重传超时就进行重传，提高了传输效率。

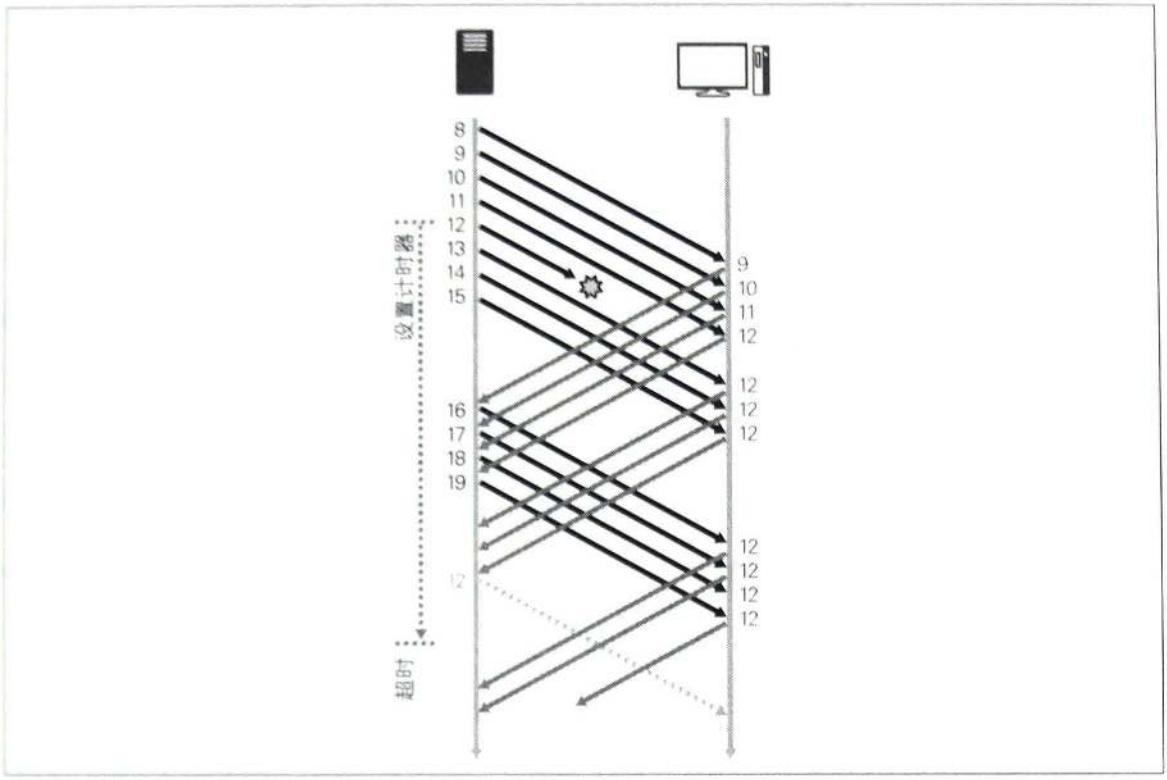

图 3.14 快速重传的流程示例

拥塞避免算法与重传控制综合影响下的流程及拥塞窗口大小的变化情况

图 3.15 展示了在拥塞避免算法与重传控制的综合影响下，拥塞窗口大小 *cwnd* 的变化情况。

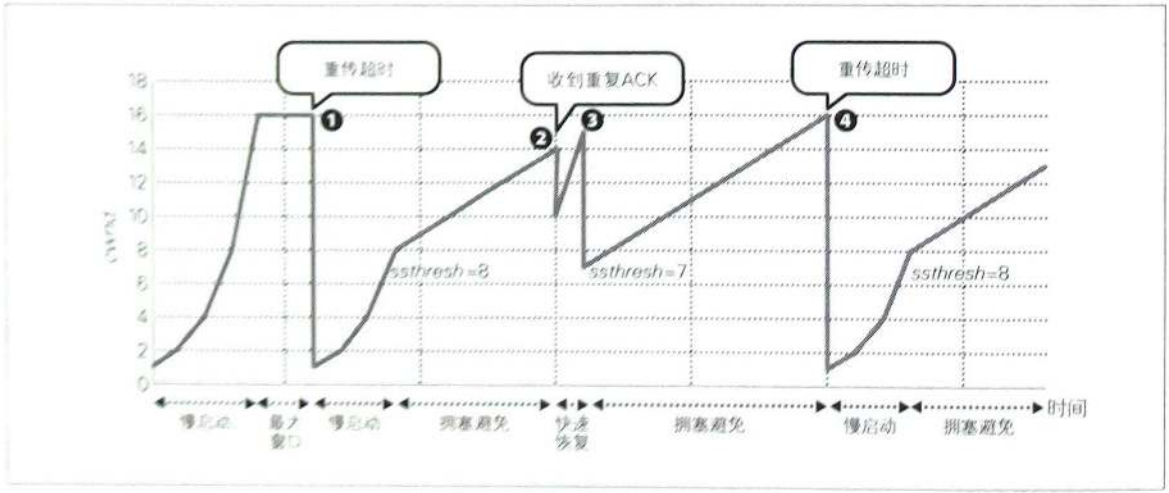

图 3.15 拥塞窗口的变化

在发送方刚开始发送数据时，使用慢启动算法，当 *cwnd* 以指数级增大到 16 时，就持续以最大值 16 发送数据。接下来，假设在图 3.15 中的时间点❶基于超时进行重传。在将 *ssthresh* 设置为当时的 *cwnd* 的一半（也就是 8）之后，进入慢启动阶段，然后等 *cwnd* 达到 *ssthresh* 后进入拥塞避免阶段。

在时间点❷，发送方由于收到重复的 ACK 而重传数据，并将 *ssthresh* 设置为 14 的一半，也就是 7。而 *cwnd* 的值取 *ssthresh* 此时的值再加上 3 的结果，也就是 10，然后进入快速恢复阶段。随后，在收到重复 ACK 的同时，暂时增大窗口大小，发送尚未发送的数据。

在时间点❸，发送方在收到与重传的 TCP 报文段对应的 ACK 后，将 *cwnd* 设置为 *ssthresh* 的值 7，进入拥塞避免阶段。之后，在时间点❹发送方再次出现重传超时，因此开始慢启动，然后进入拥塞避免阶段。

通过重复这一系列操作，TCP 实现了在拥堵网络下的高效、灵活且可靠的数据传输。

3.6 TCP 初期的代表性拥塞控制算法

Tahoe、Reno、NewReno 和 Vegas

上一节按照功能分别介绍了几种拥塞控制算法。TCP 有多个版本，这些版本各自搭载了上述的一些算法。本节将以 TCP 的初期版本为例，介绍 TCP 在从重传到拥塞控制的各个算法综合作用下的运行流程。

拥塞控制算法的变化

从前文介绍的各种内容来看，TCP 的基本思路如下。

- 在收到 ACK 之后如何增大窗口大小（正常情况下、发生拥塞时）
- 如何确保准确又迅速地进行重传
- 在检测到拥塞时，如何调整窗口大小

TCP从发展期到普及期，都是通过对以上这些问题进行研究及不断改良，才最终实现了更高的性能。在这期间，出现了以下这些算法。

- 使用慢启动、拥塞避免和快速重传算法的Tahoe
- 使用快速恢复算法的Reno
- 进一步优化了快速恢复算法的NewReno
- 使用新窗口控制思路的Vegas

掌握收发双方在时间序列上运行流程的同时，我们也要把握滑动窗口的变化情况，只有这样两手抓才能更好地理解TCP算法的具体流程。接下来，本书将对这些算法进行详细解说。

Tahoe 初期的TCP算法

Tahoe是初期的TCP算法，其所使用的算法有以下3种。

- 慢启动
- 拥塞避免
- 快速重传

下面以下页的图3.16为例，介绍一下Tahoe算法的主要流程及滑动窗口的变化情况。

首先，当发生重传时，开始由慢启动向拥塞避免状态转移。此时的拥塞窗口大小*cwnd*为4（❶）。发送方发送4个TCP报文段（11号～14号），然后在每次收到对应的ACK之后，按照拥塞避免算法增大*cwnd*。当收到4个ACK之后，*cwnd*增大到5，然后发送5个TCP报文段，也就是15号～19号报文段。

假设这个时候16号报文段丢失了。接收方已经完整地收到了15号之前的TCP报文段，所以此时会持续发送请求16号报文段的ACK（❷）。从第2个请求16号报文段的ACK开始都是重复的ACK，也就是说，由于窗口无法滑动，所以不会发送新报文段。

当收到3个重复的ACK之后（请求16号数据的ACK其实收到了4

个），发送方判断 16 号报文段已经丢失，并进行重传（❸）。与之同时，将 *cwnd* 设置为 1，从慢启动重新开始。重传开始后，发送方虽然会再次收到请求 16 号报文段的 ACK，然而由于无法滑动窗口，所以不会发送新的 TCP 报文段。

接收方在收到重传的 TCP 报文段之后，便收到了 20 号以前的数据，因此就会发送请求 21 号 TCP 报文段的 ACK。发送方在收到这个 ACK 之后，将 *cwnd* 增大 1，同时可以发送 2 个 TCP 报文段（21 号和 22 号）（❹）。随后的流程，同样是按照慢启动算法，将 *cwnd* 增大到 *ssthresh* 大小（❺）。

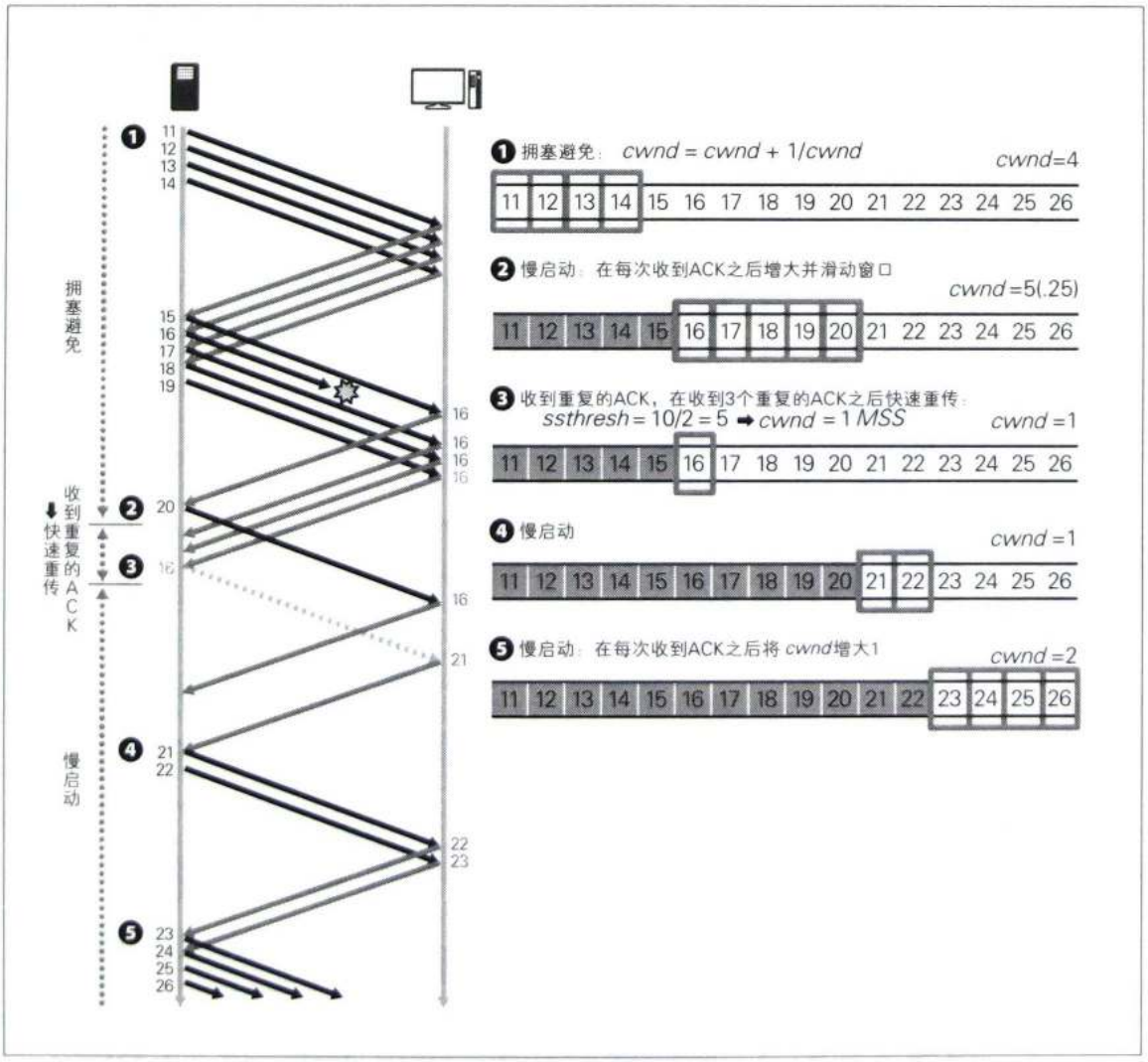

图 3.16 Tahoe 的流程示例

Reno 快速恢复

Tahoe算法在拥塞程度一般的情况下，仍然在快速重传后让*cwnd*从最小值开始增大，这种做法其实效率并不高，也就是说有改进的余地。因此Reno算法增加了快速恢复的功能。快速恢复如3.4节所述，是在快速重传之后仍然继续发送数据的一系列操作。

Reno算法的运行流程和滑动窗口的变化情况如图3.17所示。在数据刚开始发送时，从慢启动开始，当*cwnd*值达到8之后（❶），一部分TCP报文段（3号）丢失。接收方没有收到3号报文段，因此重复发送请求3号TCP报文段的ACK。此时发送方仍处于慢启动阶段，因此在每次收到ACK之后，将*cwnd*增大1，同时发送TCP报文段数据（❷）。

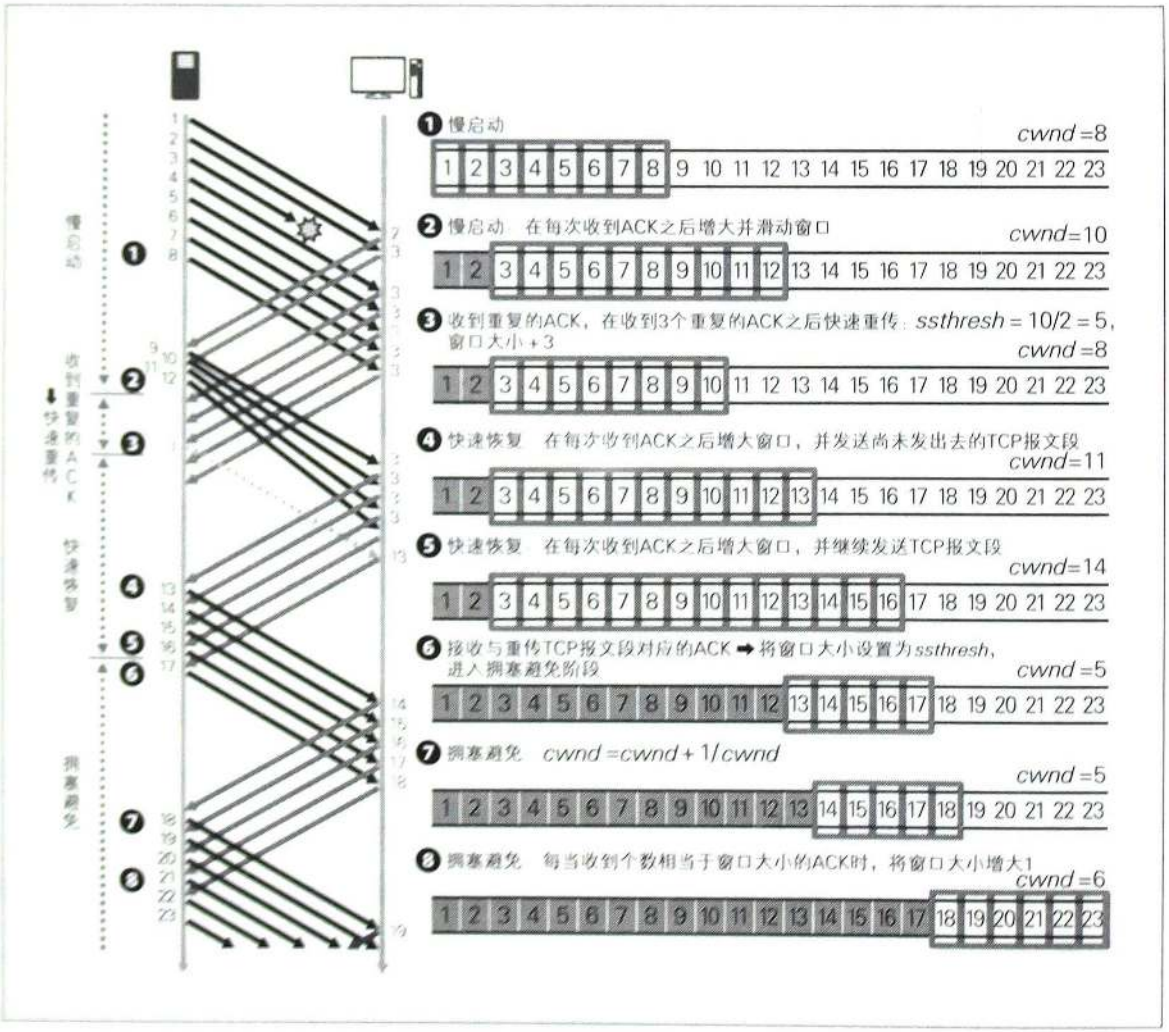

图3.17 Reno的流程示例

然后，发送方再次收到请求 3 号报文段的 ACK。此时，发送方进入接收重复 ACK 的阶段，在收到 3 个重复的 ACK 后，它判断当前 TCP 报文段已经丢失，并进行重传，即进入快速恢复阶段（❸）。此时，*cwnd* 已经增大到 10，因此将 *cwnd* 的一半加上重复 ACK 次数 3，换句话说，将 *cwnd* 设置为 5+3=8。此外，将进入拥塞避免阶段后窗口大小的一半，也就是 *ssthresh* 的大小 5 保存起来。

接下来，发送方虽然还会收到请求 3 号报文段的重复 ACK，但此时每次收到重复的 ACK 之后，就让 *cwnd* 增大 1。窗口扩大之后，只要窗口中有尚未发送的 TCP 报文段，就将这些数据发送出去（❹～❺）。此时，*cwnd* 增大到 14。

当收到重传报文段对应的 ACK 之后，就可以认为重传已经完成，此时进入拥塞避免阶段（❻）。将 *cwnd* 设为 *ssthresh* 的值（=5），然后在每次收到 ACK 之后，就一边滑动窗口一边发送 TCP 报文段（❼）。随后每当收到个数相当于 *cwnd* 大小的 ACK 后，就让窗口大小增大 1，也就是说慢慢地增加 TCP 报文段的发送数量（❽）。

Reno 中的快速恢复算法的思路是，在收到重复的 ACK 后，增大拥塞窗口大小。通过这一手段，便可以无须暂停发送数据，从而实现更高的传输效率。

然而，Reno 仍有些问题待解决。具体来说，就是在多个 TCP 报文段丢失时会出现问题。当 2 个以上的报文段丢失时，也可以在收到重复的 ACK 之后进行重传。然而，从图 3.17 可以看出来，即使重传了 3 号 TCP 报文段之后，请求 3 号报文段的 ACK 也会继续发送回来，这就导致想要收到下一个丢失报文段所对应的重复 ACK，要花费更多的额外时间，这很可能导致最终超时。

为了处理这一问题，人们设计了 NewReno，它改良了快速恢复算法。

NewReno 引入新参数（recover）

针对快速重传阶段多个 TCP 报文段丢失的情况，NewReno 算法进行