Stanford CS144 计算机网络（Introduction to Computer Networking）学习笔记

数据来源：斯坦福大学 CS 144《Introduction to Computer Networking》Fall 2026 公开课程材料，访问日期 2026-11。 已实际抓取并逐字阅读的材料包括：课程主页（https://cs144.github.io/）、课程大纲与评分说明（`logistics.pdf`）、Lab FAQ（`lab_faq.html`）、8 份公开讲义幻灯片（`week-1-logistics`、`week-1-reliability`、`week-2-tcp`、`week-3-packet-switching-a`、`week-4-why/what/how_congestion_control`、`routing-intro`）、Checkpoint 0–4 的 lab handout PDF（`assignments/check0.pdf` … `check4.pdf`），以及课程主页上以 Google Docs 形式发布的 11 篇 lecture notes（经 `export?format=txt` 取得正文）。 补充参考：RFC 9293（TCP）、RFC 5681（拥塞控制）、RFC 6298（RTO）、RFC 6675（SACK）、RFC 7323（窗口缩放/时间戳）、RFC 9438（CUBIC）、RFC 9000/9114（QUIC/HTTP3）。 推荐教材：Kurose & Ross《Computer Networking: A Top-Down Approach》(8th ed.)；Peterson & Davie《Computer Networks: A Systems Approach》（全文免费：https://book.systemsapproach.org/）；Kaashoek & Saltzer《Principles of Computer System Design》第 7–8 章（课程主页列为可选教材）。 本文档为个人学习笔记，内容版权归斯坦福大学及各位作者所有，仅用于个人学习目的。凡属笔者推断、或课程材料未明确说明之处，均已标注“推断”或以课程当学期主页为准。

1.1 一句话定位
1.2 课程目标（来自 Week 1 讲义《Course intro》）
1.3 先修要求与工作量（来自 `logistics.pdf`）
1.4 评分构成（Fall 2026 主页 Grading Weights，逐项照抄）
1.5 考试与日程（Fall 2026）
1.6 五个公开的 Checkpoint（check0–check4）
1.6.1 五个 checkpoint 的依赖关系
1.6.2 关于“课程全貌”的一句备注
1.7 课程周主题地图
1.8 学术诚信：一条必须知道的“红线”
1.9 建议的学习路径
2.1 动机：为什么需要“服务抽象”这层窗户纸
2.2 封装（Encapsulation）：一层套一层的信封
2.3 复用（Multiplexing）：用一个资源跑很多件事
2.4 幂等（Idempotency）：对付“重复投递”的唯一正解
2.5 服务抽象的堆叠图
2.6 常见误解
3.1 两种网络：电路交换 vs 分组交换
3.2 三种时延：传播、传输（序列化）、排队
3.3 端到端时延：store-and-forward 的串联
3.4 排队：确定性模型与“平均队长”算法
3.5 真实世界的时延分布：CDF 与抖动
3.6 带宽时延积（BDP）：管道里到底能装多少水
3.7 常见误解
4.1 什么叫“可靠”
4.2 最小可靠传输：停等（stop-and-wait）
4.2.1 三件套
4.2.2 为什么必须有序号：重复与“重复的 ACK”
4.2.3 停等协议的效率：算出来才知道有多差
4.3 “两个将军问题”与不完美可靠性
4.4 从“可靠数据报”到“可靠字节流”的两级台阶
4.5 常见误解
5.1 滑动窗口：把“等一个”变成“等一批”
5.2 累计确认（cumulative ACK）：一句话胜过一堆 ACK
5.3 Go-Back-N 与 Selective Repeat
5.4 超时重传：RTO 该怎么算
5.4.1 Jacobson/Karels 算法（RFC 6298）
5.4.2 Karn 算法：重传过的段不能用来测 RTT

5.4.3 CS144 lab 采用的简化规则（check3 原文）
5.5 快速重传（fast retransmit）与快速恢复
5.5.1 为什么需要它
5.5.2 快速恢复（fast recovery）的窗口操作
5.5.3 SACK：告诉发送方“我到底缺哪几段”
5.6 一个完整的手算例子
5.7 常见误解
6.1 为什么端点必须自己校验
6.2 Internet 校验和（RFC 1071）：反码求和
6.3 校验和能查出什么、查不出什么
6.4 UDP 校验和的“可选性”与 IPv6 的变化
6.5 常见误解
7.1 为什么把 TCP 头拆到比特级
7.2 TCP 头逐字段表（依据 RFC 9293 §3.1 的图 1）
7.3 三种序号：seqno / absolute seqno / stream index
7.4 关键选项（options）
7.5 为什么 ISN 要随机
7.6 常见误解
8.1 从“可靠字节流”到“连接”：为什么需要握手
8.2 同时打开（simultaneous open）：不是“经典三次握手”但仍合法
8.3 四次挥手：为什么关闭要四步
8.4 状态机（RFC 9293 图 5 的精简版）
8.5 TIME_WAIT 为什么必须存在
8.6 RST：不体面的结束方式
8.7 常见误解
9.1 MSS 与 MTU：为什么是 1460
9.2 Nagle 算法（RFC 896）：小包合并
9.3 延迟 ACK（delayed ACK，RFC 1122）
9.4 糊涂窗口综合征（Silly Window Syndrome, SWS）
9.5 Keepalive：一条有争议的机制
9.6 常见误解
10.1 两个窗口，一个上限
10.2 自时钟（self-clocking）：为什么用“窗口”而不是“速率”
10.3 发送方眼里，包可能在哪
10.4 窗口太小或太大分别会怎样
10.5 拥塞崩溃（congestion collapse）与公平性
10.6 拥塞信号：三种可选输入
10.7 常见误解
11.1 AIMD：一句话算法
11.2 锯齿图：AIMD 的稳态
11.3 慢启动（slow start）：名字骗人，其实是指数增长
11.4 为什么 AIMD 能收敛到公平（一个漂亮的小证明）
11.5 AIMD 的吞吐公式（Mathis 公式）
11.6 Tahoe、Reno、NewReno
11.7 超时 vs 快速重传的窗口处理对照
11.8 常见误解
12.1 为什么 Reno 不够用
12.2 CUBIC（RFC 9438）：用三次函数替代线性
12.3 BBR：不用丢包，用“带宽 × 时延”建模
12.4 ECN 与 AQM：让路由器“提前说话”
12.5 一张总表：五代拥塞控制
12.6 常见误解
13.1 转发（forwarding）与路由（routing）的区别
13.2 前缀（prefix）与 CIDR
13.3 最长前缀匹配（Longest Prefix Match, LPM）
13.4 如何快速做 LPM：三种数据结构
13.5 转发表里还有什么
13.6 常见误解
14.1 自治系统（AS）：把 Internet 切成“政策域”
14.2 OSPF：链路状态 + Dijkstra
14.3 BGP：路径向量（path vector）与政策
14.3.1 导入与导出政策（这是 BGP 的灵魂）
14.3.2 BGP 的属性与选路顺序
14.3.3 iBGP 的两种拓扑
14.4 BGP 的脆弱性：没有认证的全球协议
14.5 Internet 的“商业地理”：从 1995 到今天的扁平化
14.6 常见误解
15.1 默认排队：FIFO（先来先服务）
15.2 严格优先级（strict priority）
15.3 公平队列（FQ）与加权公平队列（WFQ）
15.3.1 从“按包轮转”到“逐比特轮转”
15.3.2 加权版本
15.4 令牌桶（token bucket / leaky bucket）：限制突发
15.5 常见误解
16.1 “The eras tour of home networking”：九个层级
16.2 NAPT 的规则表

16.3 四种 NAT 映射类型（RFC 3489 的分类）
16.4 P2P 穿透的四种方案
16.5 NAT 破坏了什么
16.6 常见误解
17.1 应用层要的四种“可靠性语义”
17.2 HTTP/1.1：文本协议与它的三个瓶颈
17.3 HTTP/2：二进制分帧 + 多路复用
17.4 QUIC 与 HTTP/3：把可靠传输搬进用户态
17.5 DNS：Internet 的“电话簿”
17.6 往下走一层：物理层与弹性缓冲（Elastic Buffers）
17.7 常见误解
18.1 总览：一条单向依赖链
18.2 check0：ByteStream 与 webget
18.3 check1：Reassembler
18.4 check2：Wrap32 与 TCPReceiver
18.5 check3：TCPSender（全 lab 最难）
18.6 check4：测量真实世界
18.7 五个 checkpoint 的共同“元技能”
20.1 必须能“不看书画出来”的 10 张图
20.2 自测题（12 题，答案在后）
20.3 答案
20.4 考前 3 小时清单
21.1 课程官方材料（均已实际抓取，访问日期 2026-11）
21.2 Google Docs 形式的 lecture notes（经 `export?format=txt` 取得正文）
21.3 教材与标准
21.4 未能获取 / 存在不确定性的材料

第 1 章 课程概览：这门课到底在教什么

1.1 一句话定位

CS144 是一门**“动手把 Internet 拆开再装回去”**的系统课：它不只讲协议长什么样，而是让你用 C++ 亲手实现协议栈里最关键的几块——可靠字节流、TCP 收发两端、网络接口、IP 路由器——然后用自己写的 TCP 去和 Linux 内核的 TCP、和真实服务器对话。

课程主页（Fall 2026）对 lab 的描述是：在这套 lab 里你会构建“Internet 的相当一部分：一个路由器、一个网络接口，以及 TCP 协议（把不可靠数据报变成可靠字节流）”。lecture notes 里则反复强调同一句话的另一种说法：

> 网络里几乎所有有趣的问题，都是**在一个不可靠的服务抽象之上提供另一个更可靠的抽象**。

1.2 课程目标（来自 Week 1 讲义《Course intro》）

#	目标	对应载体
1	理解 Internet 如何工作	分层、封装、复用、TCP/IP
2	理解 Internet **为什么这样设计**	端到端原则、尽力而为、自时钟、商业关系
3	学会**使用** Internet（工具层面）	telnet/netcat/tcpdump/Wireshark/ping/mtr
4	**构建** Internet 的重要部件	Checkpoint 0–7

#	目标	对应载体
5	掌握网络底层原理与技术	排队论、拥塞控制、路由、物理层

第 2 条最容易被忽视：课程里大量篇幅在回答“为什么不直接让网络保证可靠/保证带宽”。答案往往不是技术不可行，而是**规模、异构与激励**（BGP 那一讲干脆说“Internet 路由只有偶然地与转发分组有关，它主要是关于钱”）。

1.3 先修要求与工作量（来自 `logistics.pdf`）

- **正式先修**：CS111（操作系统）。讲义明确写“CS107 不够”，理由是这门课 38% 的成绩来自 C++ 编程 lab，需要非常熟悉 C++ 与调试工具。
- **学分**：本科生必须选 4 学分；研究生可选 3 或 4 学分。4 学分对应**平均每周约 12 小时**。
- 12 小时的官方拆解：

项目	每周时长
听课（周一、周三各 80 分钟）	2.7 h
Lab session（周三晚）	2.5 h
Tutorial（周四或周五，第 2 周开始）	1 h
课后写 lab / 复习考试	6 h
合计上限	**12 h**

- **考勤**：9 次 lab session 中至少到 4 次，才拿满 participation 分；落后于 checkpoint 时也被要求去 lab session。
- 一句很实在的劝退话（原文意思）：如果 Checkpoint 0/1 的编程让你很不舒服，建议**明年再选**，先去补 CS106L 或做点自选项目。

> 与同目录《CS106L Modern C++ Notes》的衔接：CS144 的 lab 代码风格要求（不用 `new/delete`、不用裸指针、`const` 满天飞、RAII、`std::string` / `std::string_view`）几乎就是 CS106L 的“实战考场”。先刷 CS106L 的 RAII、移动语义、`std::optional`、`std::string_view` 四节，再开 check0 会顺很多。

1.4 评分构成（Fall 2026 主页 Grading Weights，逐项照抄）

类别	项目	权重
考试	期中 Midterm	20%
考试	期末 Final	20%
Lab	正确性 Correctness	20%

类别	项目	权重
Lab	主观分 Subjective（公开的 5 个 checkpoint 中取最好若干次，**最后一个必做**）	18%
Tutorial	主观分（8 次取最好 6 次，记 ✔⁺/✔/✔⁻/⌀）	3%
Tutorial	随堂小测 Quizzes（8 次取最好 6 次）	16%
其他	参与度/互助 Participation & helpfulness	3%
	合计	**100%**

几点解读：

1. **Lab 合计 38%**（20% + 18%），与 `logistics.pdf` 里“38% of your grade is based on the programming lab in C++”完全对上。
2. **考试合计 40%**，两次各 20%，期末不比期中重。
3. **“取最好 6 个”是一条重要的容错设计**：它允许你崩掉若干次而不致命（但最后一个不能崩）。课程主页评分栏的原始措辞是“best 6 out of 8 checkpoints”，说明**课程计划中的 checkpoint 总数多于公开的 5 个**；本文档只讨论公开可核验的 check0–check4（见 1.6 节）。
4. `logistics.pdf` 提到本学期**尝试用自动评分器来评 lab 的“主观分”**，并承诺：结果透明、可复现（明确说明不是外部 LLM 服务）、可向讲师申诉。这一点值得记住——写 writeup 时要假设评分标准是机械可检验的（结构清晰、有取舍论证、有测量数据）。

1.5 考试与日程（Fall 2026）

项目	时间	地点
期中	10 月 28 日（周三）19:30–21:00，占用第 6 周 lab session 时段	CEMEX Auditorium
期末	12 月 9 日（周三）15:30–17:00	Gates B1
讲课	周一、周三 13:30–14:50	Gates B1
Lab session	周三 19:30–22:00	CoDa B80
Tutorial	周四或周五，第 2 周（10 月 1–2 日）开始；**第 6 周期中周停一次**（10 月 29–30 日）	报名后分配

考试形式（Week 1 讲义）：闭卷、闭笔记、闭电脑；但允许带**手写/自制**的纸质“cheat sheet”——期中 1 张双面 8.5″×11″，期末 2 张双面。这条对复习策略影响很大：本笔记第 21 章的“一页纸公式表”就是为这张纸准备的。

课程主页另提供 4 套练习题与答案（2021 Fall 期中/期末、2023 Spring 期中/期末），路径形如 `exams/sp23_midterm-ans.pdf`。**练习题是这个课最值钱的复习材料**，因为它揭示了出题口味：大量“给定窗口/序号，问接收方下一步 ackno”“给定 cwnd 曲线，问发生了几次超时”这类**可手算**的题。

1.6 五个公开的 Checkpoint（check0—check4）

截至 2026-11-01，课程站 **Labs 一栏只列出 5 个 checkpoint**，对应 `assignments/check0.pdf` … `assignments/check4.pdf`，全部可正常下载（HTTP 200）；`check5.pdf` 及之后的路径返回 **HTTP 404**。课程主页 HTML 里确实还留着更靠后的 checkpoint 标题（被 `<!-- -->` 注释掉），但**没有对应的 handout、没有接口说明、没有评分点**，因此本文档不把它们当作可学习材料，只在 1.6.2 节作为“课程全貌”的一句备注。

#	标题	主题	关键交付物	建议耗时（handout 口径/经验）
0	networking warmup	环境搭建、手工 HTTP/SMTP、`webget`、内存版 ByteStream	`apps/webget.cc`、`src/byte_stream.{hh,cc}`	handout 说 2—6 小时
1	stitching substrings into a byte stream	Reassembler：把乱序/重复/重叠的子串拼回字节流	`src/reassembler.{hh,cc}` + 抓包实验报告	参考解约 87 行，handout 估 50—60 行新增
2	the TCP receiver	32 位序号 ↔ 64 位绝对序号换算、ackno 与窗口通告	`src/wrapping_integers.cc`、`src/tcp_receiver.{hh,cc}`	handout 估约 15 行（不含 Wrap32）
3	the TCP sender	窗口填充、超时重传、RTO 指数退避、与真实 Linux TCP 互通	`src/tcp_sender.{hh,cc}`、改造后的 `apps/webget.cc`	全 lab 最难的一周
4	measuring the real world	两条真实路径的长期测量（RTT/丢包/自相关/CDF/吞吐）	一份 PDF 报告（Gradescope 提交）	需要 ≥1 小时的真实测量

1.6.1 五个 checkpoint 的依赖关系

```
check0  ByteStream（内存版可靠字节流 + 流控）
   │        └── webget：用内核 TCP 取网页（理解"上层要什么"）
   ▼
check1  Reassembler（乱序子串 → 有序字节流，复用 ByteStream 的 Writer）
   │
   ▼
check2  TCPReceiver（Wrap32 序号换算 + ackno + window，复用 Reassembler）
   │
   ▼
check3  TCPSender（窗口填充 + 定时器 + 重传，复用 ByteStream 的 Reader）
   │        └── 与 Linux TCP 互通（`tcp_ipv4` ↔ `tcp_native`），1 MB challenge
   ▼
check4  真实网络测量（不再用你的 TCP，转向"测量方法论"）
```

这条链是单向依赖的：check2 的 `TCPReceiver` 内部持有 check1 的 `Reassembler`，check1 的 `Reassembler` 内部写 check0 的 `ByteStream::Writer`。所以 handout 反复强调“很难跳过某个 checkpoint”。

1.6.2 关于“课程全貌”的一句备注

课程主页评分栏写的是“best 6 out of 8 checkpoints”，`check0.pdf` 开头也自称“this 8-part lab assignment”，Week 1 的课程介绍幻灯片（`week-1-logistics.pdf`）则说“一个大作业，含 8 个 checkpoint（#0 到 #7）”。**这三处措辞与“只公开 5 个 handout”并不矛盾**：课程计划里有更靠后的部件（网络接口、IP 路由器、把各部件拼成一个小 Internet），但它们在本学期没有公开材料。本文档的所有实现细节、接口、评分点**只覆盖 check0–check4**；凡涉及更高层部件的段落都会标“推断”。

1.7 课程周主题地图

课程按“周单元”组织。Week 1 讲义给出的单元序列是：**(1) 基本原则 → (2) 传输 → (3) 分组交换 → (4) 拥塞控制 → (5) 实用的 Internet → (6) 应用 → (7) 路由**。主页 Lecture Notes 表格实际列出的讲次更细：

周	讲次	主题	本笔记对应章节
1	周三	Course intro	第 1 章
2	周一	Datagrams, encapsulation, and multiplexing	第 2 章
2	周三	Reliability from unreliability	第 4、5 章
3	周一	TCP I（TCP 的由来、握手、窗口）	第 7、8 章
4	周一	TCP Particulars	第 9 章
4	周三	Packet Switching I（时延、排队、WFQ）	第 3、15 章
5	周一	Why / What Congestion Control?	第 10 章
5	周三	How Congestion Control（AIMD、慢启动）	第 11 章
5	周三/五	Routing（前缀、BGP、AS 经济学）	第 13、14 章
6	周五	The eras tour of home networking	第 16 章
7	周五	The eras tour of home networking II（NAPT 逐级）	第 16 章
8	周一	Elastic Buffers	第 18 章
10	周一/周三	Internet security history / Internet security	第 17.6 节

1.8 学术诚信：一条必须知道的“红线”

`check0.pdf` 与 `logistics.pdf` 都明确写了：**ChatGPT、Copilot、Claude Code 等 AI 工具，在本课程中等同于“一个以前上过这门课的学生”**。也就是说：

- 看 AI 给出的 lab 代码 = 看往年学生的解答 = 违规；
- 与 AI 讨论“作业要求是什么”“这个 API 什么语义” = 允许（等同于讨论 handout）；
- 引用网上找到的思路，必须在源码注释里写 URL。

这条对自学者同样有指导意义：**要拿这份笔记去对照自己的实现，而不是把别人的实现贴进来**。本文档刻意只写数据结构选型、状态机与不变量、易错点，不提供可直接粘贴的完整解答。

1.9 建议的学习路径

```
第 0 周  ├─ CS106L 复习：RAII / string_view / optional / 移动语义
         ├─ 装 Ubuntu 26.04 VM（UTM 或 VirtualBox），装 git/cmake/gdb/tcpdump/tshark
         └─ 读 check0.pdf 全文 → 写 webget（约 8 行）→ 写 ByteStream
第 1 周  ├─ 读第 2、4 章（封装、可靠性）+ 写 Reassembler
         └─ 做 check1 的抓包实验（WireGuard 私网 + tcpdump + Wireshark）
第 2 周  ├─ 读第 7 章（序号空间）→ Wrap32 → TCPReceiver
         └─ 同时开始背 TCP 头字段（第 7.2 节表格）
第 3 周  ├─ 读第 8、9 章（握手/挥手、Nagle、延迟 ACK）
         └─ 写 TCPSender（最难的一周：定时器 + 重传 + 零窗口）
第 4 周  ├─ 与 Linux TCP 互通、1 MB challenge、webget 换自己的 TCP
         └─ 开始 check4 的长跑测量（要跑 ≥1 小时，越早开始越好）
第 5–6 周├─ 第 10–12 章（拥塞控制）+ 第 13–15 章（路由、排队）
         └─ 刷 4 套历年题，整理一张双面 cheat sheet
第 7–9 周├─ 第 16–18 章（NAT、应用层、物理层）
         └─ 若课程后续公开更多 checkpoint：网络接口 → 路由器 → 拼 Internet（推断）
```

第 2 章 分层、封装与复用：三个词撑起整个 Internet

2.1 动机：为什么需要“服务抽象”这层窗户纸

Week 2 周一的 lecture notes 一上来给了三个“根本概念”：**Datagram（数据报）**、**Encapsulation（封装）**、**Multiplexing（复用）**。这三个词之所以根本，是因为它们回答了“一个由几万个互不隶属的网络拼起来的东西，怎么可能工作”。

先看网络**唯一**承诺的服务：**尽力而为（best effort）地投递数据报**。notes 里用寄明信片打比方：



你把它丢进“抽象有意义”的任意一点，网络负责往目的地送

“尽力而为”到底有多不保证？讲义列了一张“可能出什么事”的清单，这张清单是整门课所有机制的**需求来源**：

故障	后果	谁来补救（对应机制）
完全丢失	数据没到	超时 + 重传（第 4 章）
重复投递 n>1 次	动作被执行多次	幂等化 / 序列号去重（2.4 节）

故障	后果	谁来补救（对应机制）
内容被篡改	数据错了	校验和（第 6 章）或加密认证
被截断	数据不完整	长度字段 + 校验和
乱序到达（后发先至）	顺序错	序列号 + 重组（第 5 章）
送到错误目的地	隐私/串线	地址校验（“best effort”允许发生）
很久以后才到	陈旧数据	TTL、序号随机化、TIME_WAIT（第 8 章）

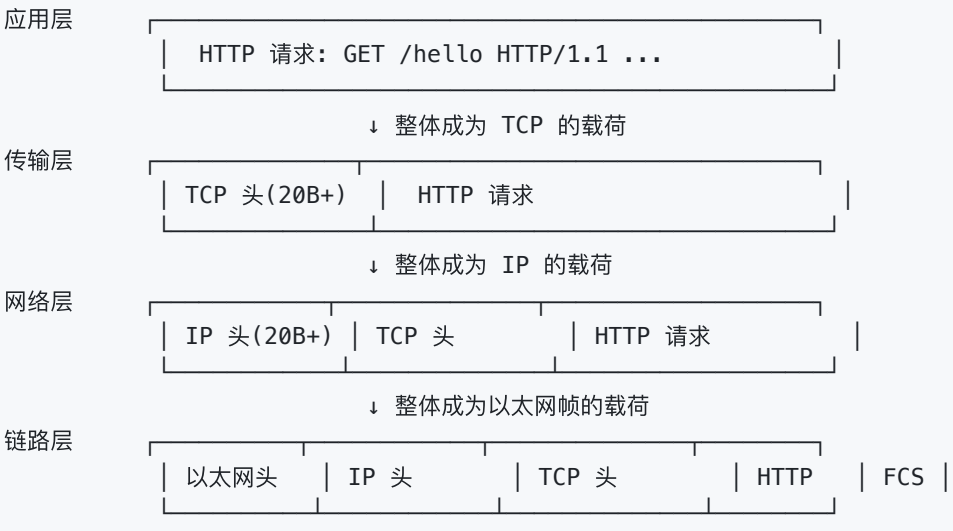
> 关键心态：**不要问“网络为什么不保证”，要问“在不保证的前提下，我能在端点上重建什么保证，代价是多少”**。TCP、QUIC、RTP、DNS 的重试逻辑，全都是这张表的不同权衡。

2.2 封装（Encapsulation）：一层套一层的信封

讲义的定义很精确：

> **封装**：把一个对象放进另一个对象里当作载荷，而外层对象对载荷的含义与内容**一无所知**。

三个类比（都来自 notes）：邮政替你运包裹但不拆开、货轮运集装箱但不看里面、Internet 运数据报但不看 payload。



“拆开一层”就是解封装（decapsulation）。每一层只读自己那一层的头，然后把载荷交给上一层——这就是为什么“路由器不需要懂 HTTP”“交换机不需要懂 IP”（第 16 章会看到交换机只看以太网头）。

这个设计带来一个非常好用的工程性质：**中间设备可以只实现自己那一层**。也带来一个副作用：**中间设备可以只看头就作恶**（防火墙、NAT、流量识别，见第 16 章）。

2.3 复用（Multiplexing）：用一个资源跑很多件事

讲义定义：**复用 = 通过运行期标识符（runtime identifier）分派，从而共享一个资源。**

```
一条链路（资源）
   ├── 按“目的 IP 地址”分派          → 不同主机的数据报共享这条链路        【网络层复用】
   ├── 按“IP 头里的 protocol 字段”分派 → 不同传输协议共享主机间投递能力      【协议复用】
   │        protocol = 1  → ICMP (ping)
   │        protocol = 6  → TCP
   │        protocol = 17 → UDP
   └── 按“目的端口号”分派             → 不同进程共享主机间数据报能力        【进程复用】
            port = 53  → DNS
            port = 80  → HTTP
            port = 443 → HTTPS
```

三层复用叠起来，才是“一台机器上同时开 200 个 Chrome 标签页”能工作的原因。**五元组** `（协议，源IP，源端口，目的IP，目的端口）` 就是这套复用体系的“完整地址”。

配套的系统知识（notes 里提到、也值得记住）：

文件	内容	例子
`/etc/protocols`	IP protocol 编号 → 名字	`tcp 6`、`udp 17`、`icmp 1`
`/etc/services`	端口号 → 服务名	`http 80/tcp`、`domain 53/udp`

> 一个常被问的问题：**Firefox 怎么知道该用哪个源端口？** 答案：内核在 `connect()` 时分配一个不与现有连接冲突的临时端口（ephemeral port），通常落在 `/proc/sys/net/ipv4/ip_local_port_range` 指定的范围里。

2.4 幂等（Idempotency）：对付“重复投递”的唯一正解

这一节是 Week 2 周三 notes 里最精彩的一段推理，也是理解 TCP 设计动机的钥匙。

面对“可能重复投递”，讲义先把操作分成两类：

类别	例子	重复执行的后果	对策
幂等（idempotent）	`get(key)`、`set(door=open)`、`GET /index.html`	最终状态不变	直接重传即可
非幂等	`pop(7)`、`push("hi")`、`launch_missile()`、`POST /billpay`	状态被改多次，出错	需要**去重标识**

讲义给了一个漂亮的工程技巧：**把非幂等操作变幂等**——给每次调用一个唯一 ID，服务端维护“已执行过的 ID 集合”：

```
// 讲义中的"发射导弹"示例（伪代码复述）
void recv(const string& service) {
    unordered_set<uint64_t> launched;      // 已执行过的请求 ID
    UDPSocket sock; sock.bind(Address("0", service));
    Address source("0"); string payload;
    while (true) {
        sock.recv(source, payload);
        if (payload == "launch_one_missile" + id) {
            if (!launched.contains(id)) { launch_missile(); launched.insert(id); }
            sock.sendto(source, "ack");     // 无论如何都回 ack
        }
    }
}
```

代价是**服务端要记住所有见过的 ID**——这正是 TCP 用“序列号 + 累计确认”而不是“请求 ID 集合”的原因：序列号天然有界（滑动窗口），不需要无限记忆。

接着讲义一路把 ByteStream 的 `push/pop` 推导成 TCP 的三元组：

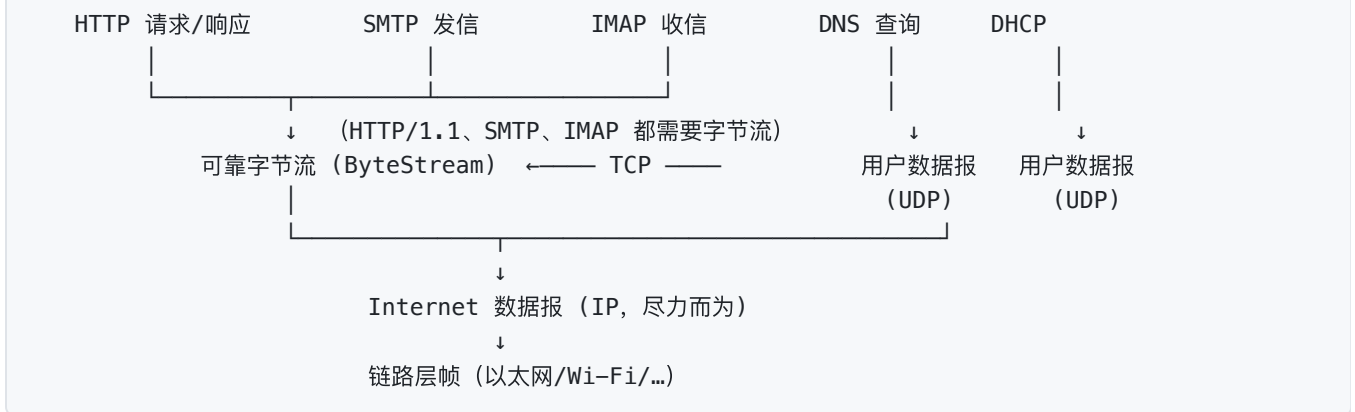
```
想让 push("abcd") 幂等，消息里必须有：
   ① first_index（这份数据在流里的位置）→ 位置相同即同一份数据
   ② data（数据本身）
   ③ FIN（流是否到此结束）
   ⇒ 这就是 TCPSenderMessage { seqno, SYN, payload, FIN }

想让发送方知道"要不要重发"，接收方必须回：
   ① ackno（我下一个需要的序号）→ 累计确认
   ② window_size（我还愿意收多少）→ 流控
   ⇒ 这就是 TCPReceiverMessage { ackno, window_size }
```

这段话把“为什么 TCP 头长这样”讲透了：**TCP 头不是委员会拍脑袋定的，它是“让字节流的写入变成幂等操作”的最小信息集**。

2.5 服务抽象的堆叠图

notes 里给了一张“谁架在谁上面”的清单，非常值得抄下来当心智地图：



2.6 常见误解

误解	事实
“IP 是可靠传输，丢了会重发”	IP **只**做尽力而为；重发是 TCP 的事。IP 头里的校验和只保护**头**，不保护数据。
“UDP 不可靠，所以不能用 UDP 做可靠传输”	可靠性是**端点**的责任。QUIC 就在 UDP 上重建了可靠传输（第 17 章）。
“分层是硬性规定”	分层是**设计约定**，不是法律。NAT、隧道、MPLS、VPN 都在“违反”分层，只是它们付得起代价（第 16 章）。
“封装会浪费带宽，所以层越少越好”	每层头 20–40 字节，相对 1500 字节 MTU 是 2–5%；换来的是互操作性。真正的浪费在**重传**与**排队**。
“端口是进程的属性”	端口是**内核**分配的复用标识；同一进程可以有多个端口，同一端口也可以被多个连接共享（由四元组区分）。

第 3 章 分组交换与时延：把“快不快”算出来

3.1 两种网络：电路交换 vs 分组交换

Week 4 周三《Packet Switching I》从电话网讲起：

- **电路交换（circuit switching）**：每通电话在物理上占用一条端到端的电路，中间交换局由人工（后来是电子）把线接起来。优点是**速率稳定、时延固定**；缺点是**建立/拆除昂贵**，而且“传输期间独占资源”——对“只发一小段数据”的 Web 请求来说是巨大浪费。
- **分组交换（packet switching）**：把数据切成小片（分组），每个分组自带目的地址，路由器**逐跳转发**。链路被所有流**统计复用**（statistical multiplexing），没有建立开销，代价是**排队与丢包**。

这张对比表要背下来：

维度	电路交换	分组交换
资源占用	通话期间独占	按需共享
建立开销	有（信令）	无
时延特性	固定、可预测	随负载波动（排队时延）
突发流量	浪费（空闲也占着）	高效
故障影响	电路断则通话断	可绕路（路由协议）
计费直觉	按时间/距离	按流量

3.2 三种时延：传播、传输（序列化）、排队

这是全课最基础的一组公式，务必分清 **propagation（传播）** 与 **transmission/serialization（传输/序列化）**——这两个词在中文里都被译成“传输”，是最大的混淆源。

```
发送端                                                    接收端
  │  ═══════════════════════════════════════════════════  │
  │        ←──────────── 链路长度 l，速率 r ──────────→    │
  │                                                       │
  │  传播时延 t_l = l / c          （一个比特在介质里跑完全程的时间）
  │  传输时延 t_p = p / r          （把 p 比特"推上"链路所需时间）
  └─ 一个分组跨越一条链路的总时间 = t_p + t_l
```

量	符号	公式	物理含义	典型值
传播时延	t_l	l/c	光/电信号在介质中跑完距离的时间	光纤 $c \approx 2\times10^8$ m/s；1000 km ≈ 5 ms
传输时延	t_p	p/r	把整包比特“打出去”的时间	1500 B @ 1 Gb/s ≈ 12 μs
排队时延	$d(t)$	依赖负载	在路由器缓冲区里等待的时间	0 到几十 ms
处理时延	—	常数级	查表、校验	μs 级

讲义例题（原文数据）：1000 km 光纤、$c = 2\times10^8$ m/s → 单比特传播 5 ms；100 bit 的分组在 10 Mb/s 链路上 → 传输 10 μs… 注意讲义写的是“10 + 5 = 15 ms”，这里的 10 是 **10 μs 量级被写成 ms 的口径差异**（幻灯片用的是“100 bit / 10 Mb/s = 10 μs”，而例子给的是 15 ms，说明该例实际取的是不同的 p/r 组合）。**做作业时以公式为准，不要背例子里的数字**。

3.3 端到端时延：store-and-forward 的串联

多跳路径上，每个交换机必须**收完整包再转发**（store-and-forward），于是：

```
A ──(l1,r1)── S1 ──(l2,r2)── S2 ──(l3,r3)── S3 ──(l4,r4)── B

时间轴（横轴为时间）：
A   ├─p/r1─┤
S1          ├─p/r2─┤
S2                  ├─p/r3─┤
S3                          ├─p/r4─┤
B                                  └→ 到达
    └──l1/c──┘ （每段还有各自的传播时延，与上面的发送时间部分重叠）
```

公式（讲义原文）：

$$\tau = \sum_i \left(\frac{p}{r_i} + \frac{l_i}{c} \right)$$

加上排队：

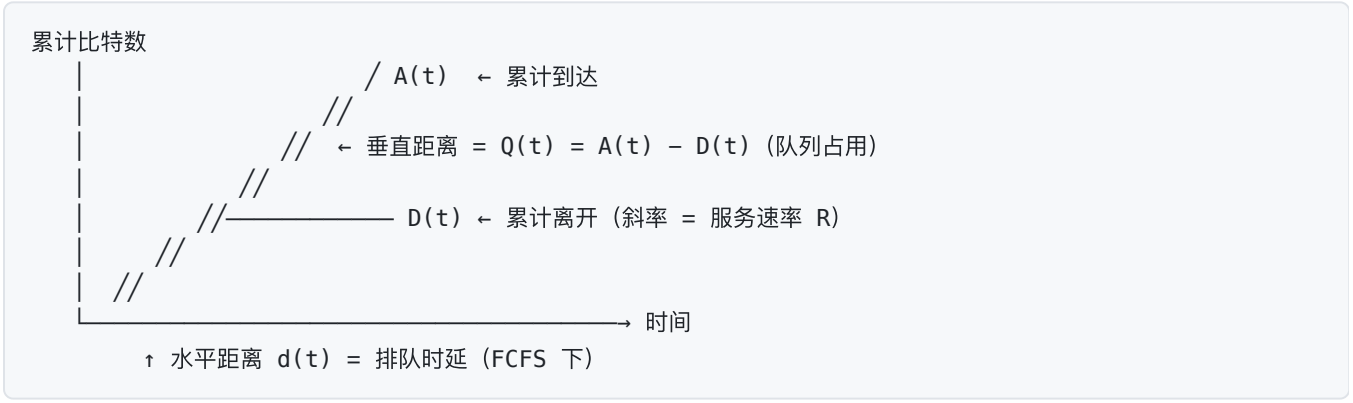
$$\tau = \sum_i \left(\frac{p}{r_i} + \frac{l_i}{c} + Q_i(t) \right)$$

三个必须内化的直觉：

1. **“跳数多”主要增加的是 p/r 与处理时延，不是 l/c**（因为传播是并行发生的）。
2. **“带宽大”不会让传播时延变小**。跨太平洋 RTT 下限约 100 ms，是光速决定的，跟 1 Gb/s 还是 100 Gb/s 无关。
3. **排队时延是唯一“可以被算法改变”的部分**——拥塞控制（第 10–12 章）与排队规则（第 15 章）都在动这一项。

3.4 排队：确定性模型与“平均队长”算法

讲义用累计曲线（cumulative arrivals/departures）建模一个 FIFO 队列：



性质：$A(t)$、$D(t)$ 单调不减，且 $A(t) \geq D(t)$。

讲义例题 1（store-and-forward）：每秒有一个 500 bit 的分组到达，到达速率 10 000 b/s（即 0.05 s 内到完），队列服务速率 1 000 b/s。求平均队长。

- 先以 10 000 b/s 填 0.05 s，积累 500 bit；
- 再以 1 000 b/s 排空，需要 0.5 s；
- 前 0.55 s 内平均队长 = 250 bit；
- 每周期有 0.45 s 队列为空；
- 平均队长 $= 0.55 \times 250 + 0.45 \times 0 = 137.5$ bit。

讲义例题 2（cut-through，直通转发）：同样的参数，但交换机边收边发（收到就能转发一部分）。

- 0.05 s 内队列只积累到 $500 - 50 = 450$ bit；
- 再用 0.45 s 排空；
- 前 0.5 s 平均队长 225 bit；每周期 0.5 s 空；
- 平均队长 $= 0.5 \times 225 + 0.5 \times 0 = 112.5$ bit。

> **考点**：这两个数字的差别（137.5 vs 112.5）说明**“直通转发”能把平均排队时延降低约 18%**，代价是交换机在收到完整包与校验和之前就开始转发（可能转发一个坏包）。

3.5 真实世界的时延分布：CDF 与抖动

讲义给了一张两条路径的 RTT CDF 对比图（数据来自课程幻灯片）：

路径	距离	RTT 变化幅度
Stanford ↔ Princeton	约 4000 km	抖动约 50 ms
Stanford ↔ Tsinghua	约 10 000 km	抖动约 200 ms

读法：CDF 曲线越“陡”，时延越稳定；曲线在尾部拖得很长，说明有排队/重路由发生。**抖动（jitter）随时延量级同比放大**，这对实时音视频（需要 jitter buffer）是硬约束。

3.6 带宽时延积（BDP）：管道里到底能装多少水

$$BDP = \text{瓶颈链路速率} \times RTT$$

讲义例子：5 Mb/s × 100 ms = **62.5 KB**（注意是字节，$5 \times 10^6 \times 0.1/8 = 62500$ B）。

这张“管道图”值得画在 cheat sheet 上：

```
          瓶颈链路 5 Mb/s
发送端 ═══════════════════════════════▶ 接收端
       |←──── RTT = 100 ms ────→|

管道"容量" = 5 Mb/s × 100 ms = 62.5 KB
若在途数据 < BDP：链路有空转（吞吐上不去）
若在途数据 = BDP：链路满、队列空（理想）
若在途数据 > BDP：多出来的部分堆在瓶颈队列里 → 排队时延↑，再涨就丢包
```

“无丢包窗口” = BDP + 路由器最大队列长度。这个量在第 10–12 章会反复出现：**所有拥塞控制算法本质上都在猜这个值**。

3.7 常见误解

误解	事实
“带宽越大，时延越小”	带宽只影响 p/r；跨洋 RTT 由光速决定。所谓“低时延专线”卖的是**排队**与**绕路**，不是光速。
“传播时延和传输时延是一回事”	前者是距离/光速，后者是包长/速率。1 GB 的文件在 1 Gb/s 上传输要 8 s，但传播仍是 ms 级。
“ping 的 RTT 就是网络时延”	它还包含**端点处理**、**中断合并**、**NIC 排队**，在虚拟化环境里可能被放大数倍。check4 让你量 1 小时就是为了看分布而不是单点。
“丢包一定意味着拥塞”	也可能是链路误码（无线）、路由抖动、ACL 丢包、接收端 buffer 溢出。**区分这两者是 BBR 类算法的核心难点**。

误解	事实
“平均时延重要”	对交互式应用，**尾时延（p99）** 比均值重要得多；CDF 图比均值更能说明问题。

第 4 章 可靠传输（一）：从“不可靠”长出“可靠”

4.1 什么叫“可靠”

Week 1 周三讲义《Reliability from unreliability》给了一个严谨定义：

> 一个模块是可靠的，当它：**提供某个明示的抽象/接口**，**在底层故障面前仍能满足该抽象**，并且**做不到时明确地报告失败**。

第三句最容易被忽略：**可靠 ≠ 永不失败，可靠 = 失败可见**。TCP 的重置（RST）、连接超时、`ECONNRESET`，都是“体面地报告失败”的机制。

讲义接着列举“用户真正想要的”四种东西，它们对可靠性的要求完全不同：

需求	例子	关键要求
可靠地取回一小段数据	“`cs144.keithw.org` 的 IP 是多少？”	要么拿到答案，要么超时
可靠地执行一个动作	“第 7 条消息的内容是：发射鱼雷！”	**恰好一次**（exactly once）
可靠字节流	双向、有序、无错	不丢、不重、不乱
大文件可靠传输	“附件是 20 MB 的文件”	上面所有 + 高吞吐

4.2 最小可靠传输：停等（stop-and-wait）

4.2.1 三件套

Week 3 讲义《TCP》给出最小方案：**把数据报反复发送，直到收到确认**。三件工具：

1. **确认（ACK）**：接收方告诉发送方“我收到了”。
2. **定时器与超时（timer & timeout）**：发送方等太久就重发。
3. **重传（retransmission）**：重发同一份数据。

```
发送方                                          接收方
   |──────── 数据 (seq=0) ──────────────────►|
   |◄──────── ACK (ack=1) ──────────────────|
   |                                          |
   |  若超时未收到 ACK:                        |
   |──────── 数据 (seq=0) 重传 ──────────────►|  ← 接收方可能收到两份!
   |◄──────── ACK (ack=1) ──────────────────|
```

4.2.2 为什么必须有序号：重复与“重复的 ACK”

- 接收方收到重复数据，必须**丢弃**并**再回一次 ACK**（因为上一个 ACK 可能丢了）。
- 发送方收到重复 ACK，不能认为“又成功了一次”。
- 于是需要**序号**：接收方按序号判断“这是新的还是旧的”。

停等协议下，1 bit 序号就够：因为发送方在收到 ACK 之前最多只有一个包在途，接收方只需区分“我期待的那个”与“上一个”。

4.2.3 停等协议的效率：算出来才知道有多差

设：链路速率 R、单向传播时延 d、分组长度 L。则

$$U_{\text{stop-and-wait}} = \frac{L/R}{RTT + L/R} = \frac{L/R}{2d + L/R}$$

代入“1 Gb/s、RTT 100 ms、1500 B 分组”：

$$U = \frac{12\,\mu s}{100\,ms + 12\,\mu s} \approx 0.012\%$$

这就是“必须流水线化”的全部理由。用一个数字说服自己：跨洋链路上停等协议只能跑出约 120 kb/s，而链路容量是 1 Gb/s。

4.3 “两个将军问题”与不完美可靠性

讲义（check3 的 hands-on 部分）提到 Linux 在关闭连接后会“linger”一段时间，是为降低“两个将军问题（two generals problem）”的影响。

- **问题**：在不可靠信道上，双方要通过确认达成“共同知识”（common knowledge）是不可能的——最后一个 ACK 永远可能丢。
- **对 TCP 的影响**：主动关闭方无法知道“我的最后一个 ACK 对方收到了没有”，只能**等一段足够长的时间**（TIME_WAIT）再认为连接结束（第 8 章）。
- **对应用的启示**：需要“恰好一次”语义时，必须把幂等键放在**应用层**（2.4 节的导弹 ID），因为传输层给不了。

4.4 从“可靠数据报”到“可靠字节流”的两级台阶

Week 3 讲义把问题拆成两级，这个拆法是理解 TCP 的关键：

```
第 1 级：可靠地传"一条消息"
    工具：ACK + 定时器/超时 + 重传
    难点：重复（需要序号去重）

第 2 级：可靠地传"一个大消息"
    工具：把大消息切成带序号的段 + 上面全部
    难点：乱序（需要重组）、丢失（需要选择性重传）、流量控制

第 3 级（TCP 独有）：可靠地传"两条方向相反的字节流"
    工具：序号空间 + SYN/FIN 占位 + 双向独立关闭
    难点：连接生命周期、序号回绕、TIME_WAIT
```

4.5 常见误解

误解	事实
“有 ACK 就可靠了”	ACK 本身会丢。必须配定时器，并且要能处理“ACK 到了但数据没到”（不可能）与“数据到了但 ACK 丢了”（常见）。
“重传次数越多越可靠”	重传会加剧拥塞（congestion collapse，第 10 章），所以必须有**指数退避**与**放弃阈值**。
“序号可以随便定”	序号空间必须**大于**“在途窗口 + 可能的重复”，否则新旧包无法区分（第 7.3 节的 32 位序号与 MSL 的关系）。
“超时时间设短一点反应快”	设短了会**伪重传**（spurious retransmission），浪费带宽并误判拥塞。RTO 必须由 RTT 统计量推导（5.4 节）。

第 5 章 可靠传输（二）：滑动窗口、累计确认与重传策略

5.1 滑动窗口：把“等一个”变成“等一批”

动机：停等协议把链路闲置了 99.99% 的时间。既然链路能装下 BDP 那么多数据，就让多个分组同时在途。

发送方视角的字节流（每个格子是一个字节/一段）：



窗口左边界 = 最早未确认字节（send_base / SND.UNA）
窗口右边界 = 左边界 + 窗口大小（min(rwnd, cwnd)）
收到 ACK → 左边界右移 → 窗口"滑动" → 可以再发新数据

三句话记住滑动窗口：

1. **窗口大小 = 允许在途的字节数上限**；
2. **ACK 让左边界前进**（累计确认），因此窗口自动“滑”；
3. **窗口由两端共同决定**：接收方说“我能收多少”（`rwnd`），发送方自己判断“网络能承受多少”（`cwnd`），实际用 $\mathbf{min}(rwnd, cwnd)$。

5.2 累计确认（cumulative ACK）：一句话胜过一堆 ACK

规则：ACK 号 = “我下一个需要的字节序号”。它**隐式确认了此序号之前的所有字节**。

```
接收方收到：  [0..99] [200..299] [100..199]    (乱序到达)
                              ↑ 此时 ackno = 100（不是 300！）
   因为 100..199 还没到，累计确认只能报到 100。
   直到 100..199 到达，ackno 才能一跃变成 300。
```

优点：ACK 可以合并、可以丢（后一个 ACK 覆盖前一个），实现简单，状态少。 **缺点**：**一个洞会“骗”发送方重传整个窗口**。这正是“快速重传 + SACK”要解决的问题。

Week 3 讲义里对累计确认的推导非常直白：如果每个段都单独确认，则“每个发送消息都要配一个接收消息”，开销翻倍；而“Got anything? Y/N. Next needed: #3”这种**只报下一个需要的序号**的语义，天然把多个确认合并成一个。

5.3 Go-Back-N 与 Selective Repeat

维度	Go-Back-N（回退 N）	Selective Repeat（选择重传）
接收方缓冲	不缓冲乱序段，直接丢弃	缓冲乱序段
超时后的动作	从丢失处重传**之后所有**段	只重传丢失的那一段
ACK 语义	累计确认	累计 + 单独确认（SACK）
发送方窗口	$W \le 2^k - 1$	$W \le 2^{k-1}$

维度	Go-Back-N（回退 N）	Selective Repeat（选择重传）
带宽效率	差（丢一个重传一片）	好
实现复杂度	低	高（需缓冲与定时器管理）

> TCP 实际是**两者的混合体**：ACK 是累计的（像 GBN），但发送方只重传“最早的那个未确认段”（像 SR 的简化版），并可选地使用 SACK 告知“我还缺哪些段”。

5.4 超时重传：RTO 该怎么算

5.4.1 Jacobson/Karels 算法（RFC 6298）

维护两个统计量：**平滑 RTT（SRTT）** 与 **RTT 偏差（RTTVAR）**。

第一个样本（R 为测量值）：

$$SRTT \leftarrow R, \qquad RTTVAR \leftarrow R/2, \qquad RTO \leftarrow SRTT + \max(G, 4 \cdot RTTVAR)$$

后续样本（$\alpha = 1/8$，$\beta = 1/4$）：

$$RTTVAR \leftarrow (1-\beta)\, RTTVAR + \beta\, |SRTT - R|$$

$$SRTT \leftarrow (1-\alpha)\, SRTT + \alpha\, R$$

$$RTO \leftarrow SRTT + \max(G, 4 \cdot RTTVAR)$$

其中 G 是时钟粒度。RFC 6298 还给了**上下界**：RTO 最小 1 s（SHOULD，实现常降到 200 ms），最大至少 60 s。

记忆口诀：SRTT 是“平均”，RTTVAR 是“平均绝对偏差”，RTO = 平均 + 4 倍偏差 → **约等于“均值 + 4σ”，覆盖绝大多数正常样本**。

5.4.2 Karn 算法：重传过的段不能用来测 RTT

问题：你发了一个段，超时了，重传了一次，然后收到 ACK。这个 ACK 对应的是**原始发送**还是**重传**？无法区分（retransmission ambiguity）。

Karn 算法：

1. **只要发生过重传，就不为该段采集 RTT 样本**；
2. 每次重传，**RTO 翻倍**（指数退避，exponential backoff）；
3. 直到成功传输一个**未重传过**的段，才恢复用样本更新。

> 现代实现用 **TCP Timestamps 选项（RFC 7323）** 绕开这个限制：每个段带发送时间戳，ACK 回显该时间戳，于是即使重传也能精确配对（Eifel 检测的基础）。这是“Karn 算法在工程上被软化”的例子，但考试与 lab 仍按 Karn 的简化规则走。

5.4.3 CS144 lab 采用的简化规则（check3 原文）

check3 明确说这套规则“基于 RFC 6298 建议 5.1—5.6 的简化版”：

事件	动作
发送**占用序号空间**的段（数据/SYN/FIN 任一），若定时器未运行	启动定时器，RTO 毫秒后到期
所有在途数据都被确认	停止定时器
`tick()` 时定时器到期	① 重传**序号最小**的未完全确认段；② 若窗口非零：连续重传计数 +1、**RTO 翻倍**；③ 重启定时器（用翻倍后的 RTO）
收到**推进了** ackno 的 ACK	① RTO 恢复初始值；② 若有在途数据，重启定时器；③ 连续重传计数归零
收到**未推进**的 ackno	什么都不做（不重置 RTO，不清零计数）

三个易错点：

1. **“推进了 ackno”才重置 RTO**。收到重复 ACK 就重置，会让退避失效。
2. **窗口为零时不翻倍 RTO、不计数**（否则零窗口探测会把 RTO 顶到天上）。
3. **定时器只能有一个**（the retransmission timer），不是“每段一个定时器”。这也是 TCP 与“每个包一个定时器”的教学模型最大的实现差异。

5.5 快速重传（fast retransmit）与快速恢复

5.5.1 为什么需要它

超时太慢了：RTO 至少几百毫秒。而**重复 ACK 是丢包的即时信号**——如果接收方连续收到序号 5、6、7、8，它每次都会回 ackno=100（假设 100 丢了），发送方会看到一串 **duplicate ACK**。

规则（RFC 5681）：收到 **3 个重复 ACK** 就认为该段丢失，立刻重传，不等超时。

```
发送方发出：  [100] [200] [300] [400] [500]
到达情况：    [100]  ✗   [300] [400] [500]
接收方回：    ack=200 ack=200 ack=200 ack=200
                          └────┬────┘
                       三个重复 ACK (dupACK #1,#2,#3)
                                ↓
发送方：立即重传 [200] (fast retransmit)，不等 RTO
```

5.5.2 快速恢复（fast recovery）的窗口操作

Reno 的做法：

```
进入快速重传时：
    ssthresh = max(FlightSize / 2, 2*SMSS)      ← 在途数据量的一半
    cwnd     = ssthresh + 3*SMSS                ← "膨胀"3 个 MSS（因为已离开网络 3 个包）
每收到一个额外 dupACK：
    cwnd += SMSS                                ← 继续膨胀（每个 dupACK 说明又有一个包离开了网络）
收到新数据的 ACK：
    cwnd = ssthresh                             ← 收缩回"减半"值，进入拥塞避免
超时（而非 3 dupACK）：
    ssthresh = max(FlightSize/2, 2*SMSS)
    cwnd = 1*SMSS                               ← 超时比 dupACK 严重得多
    重新进入慢启动
```

5.5.3 SACK：告诉发送方“我到底缺哪几段”

问题：累计 ACK 只能说“我缺 200”，发送方不知道 300–500 是否到了。如果 200 和 400 都丢了，GBN 式重传会连 300、500 一起重发。

SACK（RFC 2018 / RFC 6675）：在 TCP 选项里携带**已收到的块区间**：

```
SACK 选项示意（左边界, 右边界）：
   Left Edge = 300, Right Edge = 501     → 我收到了 300..500
   再配一个块：Left = 600, Right = 700    → 也收到了 600..699
   结合累计 ackno = 200：
   ⇒ 发送方精确知道：缺 200..299 与 501..599，只需重传这两段
```

机制	信息量	开销	恢复效率
累计 ACK	一个点	0	差（一洞重传一片）
累计 ACK + 3 dupACK	一个点 + 丢包信号	0	中（只快重传第一个洞）
SACK	多个区间	选项 2–34 字节	好（每 RTT 可填多个洞）
DSACK（RFC 2883）	重复收到的区间	同上	用于推断伪重传

> **考试常考**：给了累计 ackno 与 SACK 块，问“发送方下一个应该重传哪一段”。答案是**序号最小的未确认段**（不是 SACK 块之间的空洞里随便挑）。

5.6 一个完整的手算例子

设 ISN = 1000，MSS = 100，初始 cwnd = 300 B，接收方窗口 1000 B。

```
t0: 发送 [1000..1099] [1100..1199] [1200..1299]    (SYN 已消耗 1 个序号，此处简化)
t1: 收到 ackno=1100  →  累计确认到 1100，窗口左边界 = 1100
    可再发 100 B（窗口右边界 = 1100+1000 = 2100，已发到 1300，还能发 800）
    cwnd 从 300 涨到 400（慢启动：每个 ACK +1 MSS）
t2: 收到 ackno=1100,1100,1100（3 个 dupACK）→ 1100 段丢失
    ssthresh = max(在途/2, 2*MSS) = max(200/2, 200) = 200
    cwnd = 200 + 300 = 500，立刻重传 [1100..1199]
t3: 收到 ackno=2100（新数据）→ cwnd = ssthresh = 200，进入拥塞避免
```

5.7 常见误解

误解	事实
“3 个 dupACK 一定意味着丢包”	也可能是**乱序**（reordering）。真实网络里 reordering 会造成伪快速重传，所以 Linux 有 `dupthresh` 可调，并用 DSACK 检测。
“超时和快速重传的处理一样”	不一样：超时 → `cwnd=1 MSS` 重进慢启动；3 dupACK → `cwnd=ssthresh` 进快速恢复。**超时是更严重的信号**。
“窗口越大越快”	窗口超过 BDP 只会把数据堆进队列（排队时延↑），超过 BDP+队列就丢包。吞吐上限 ≈ `cwnd/RTT`。
“累计 ACK 会确认乱序到达的数据”	不会。累计 ACK 只确认**连续前缀**。乱序数据只能靠 SACK 单独告知。
“RTO 越小恢复越快”	伪重传会把有效吞吐打掉，还让拥塞窗口被误砍。RTO 要覆盖 RTT 的尾部。

第 6 章 校验和与差错检测：把“被改过”找出来

6.1 为什么端点必须自己校验

链路层（以太网 FCS/CRC）已经校验过一次了，为什么 TCP/IP 还要校验和?

1. **中间设备会改包**：NAT 改地址、路由器改 TTL、负载均衡器改端口——每一步都可能引入错误。
2. **内存/总线错误**：包在主机内部（DMA、内核缓冲区）也可能被破坏。
3. **端到端原则（end-to-end argument）**：只有端点才知道数据“应该是什么”，因此完整性检查必须由端点做；链路层校验只是**性能优化**（早丢早省）。

Week 3 讲义的说法是：UDP 有 checksum、IP 头有 header checksum，所以 TCP **不需要**担心“截断与损坏”——但这句话有前提：**校验和必须真的能查出错误**。而 Internet checksum 很弱（见 6.4）。

6.2 Internet 校验和（RFC 1071）：反码求和

算法（发送方）：

1. 把校验和字段置 0；
2. 把整个待校验范围看成 16 位字的序列；
3. 做 **16 位反码和（one's complement sum）**：相加时若产生进位，把进位加回最低位（end-around carry）；
4. 结果取反，填入校验和字段。

接收方：对所有字（含校验和字段）做同样的反码求和，结果应为 `0xFFFF`（即取反后为 0）。

```
例子（16 位字）：  0x0001, 0xf203, 0xf4f5, 0xf6f7
  0x0001 + 0xf203 = 0xf204
  0xf204 + 0xf4f5 = 0x1e6f9 → 进位回卷 → 0xe6fa
  0xe6fa + 0xf6f7 = 0x1ddf1 → 进位回卷 → 0xddf2
  取反             = 0x220d    ← 这就是校验和字段
  校验时：0x0001+0xf203+0xf4f5+0xf6f7+0x220d = 0xffff ✓
```

覆盖范围（关键细节，考试爱考）：

协议	校验和覆盖范围
IPv4 头	**仅 IP 头**（不含数据），每跳都要重算（因为 TTL 变了）
UDP	伪首部（pseudo-header）+ UDP 头 + 数据
TCP	伪首部 + TCP 头 + 数据

伪首部包含：源 IP、目的 IP、协议号、TCP/UDP 长度。它不真正传输，只是让校验和**绑定到端点上**——防止“包被送到错误主机却校验通过”。

```
IPv4 伪首部（12 字节）：
   +--------+--------+--------+--------+
   |           Source Address          |
   +--------+--------+--------+--------+
   |         Destination Address       |
   +--------+--------+--------+--------+
   |  zero  | Proto  |    TCP Length    |
   +--------+--------+--------+--------+
```

6.3 校验和能查出什么、查不出什么

错误类型	Internet checksum	CRC-32
单比特翻转	✅ 必查	✅
双比特翻转	✅（同一字内 / 多数情况）	✅

错误类型	Internet checksum	CRC-32
奇数个比特翻转	✅	✅
两个 16 位字互换	❌ **查不出**（加法可交换）	✅
插入/删除 0 字节	❌ 查不出	✅
突发错误	部分	✅ 强
计算成本	极低（纯加法）	需要移位/查表

结论：Internet checksum 是“**软件实现的、够用就好**”的选择。它在 1970 年代用软件算 CRC 太贵，于是选了这个；代价是必须靠**更强的机制**（如 TCP 的 payload 校验 + 应用层 TLS/加密认证）兜底。

6.4 UDP 校验和的“可选性”与 IPv6 的变化

- **IPv4 下 UDP 校验和是可选的**（填 0 表示“没算”），因为当年为了性能允许跳过；**IPv6 下必须计算**（IPv6 头没有校验和，链路层也不一定可靠）。
- TCP 校验和**始终必填**。

> 工程含义：写 lab 时如果你算出校验和为 0，要写成 `0xFFFF`（在 UDP 里 0 表示“未计算”，所以“计算结果恰为 0”必须用全 1 表示）。这是 RFC 768 的经典陷阱，也是很多实现被 Wireshark 报“checksum incorrect”的原因。

6.5 常见误解

误解	事实
“IP 校验和保护数据”	只保护**头**。数据完整性靠 TCP/UDP 校验和。
“校验和能防篡改”	不能。攻击者改完数据可以重算校验和。防篡改需要**加密认证**（TLS、IPsec、TCP-AO）。
“以太网 CRC 过了就不用 TCP 校验”	中间设备（NAT、代理）会重写字段；且 CRC 只在链路内有效。端到端校验不可省。
“校验和越强越好”	强校验（CRC-32C）在高速链路上需要硬件卸载；软件路径上弱校验 + 上层认证是更常见的组合。

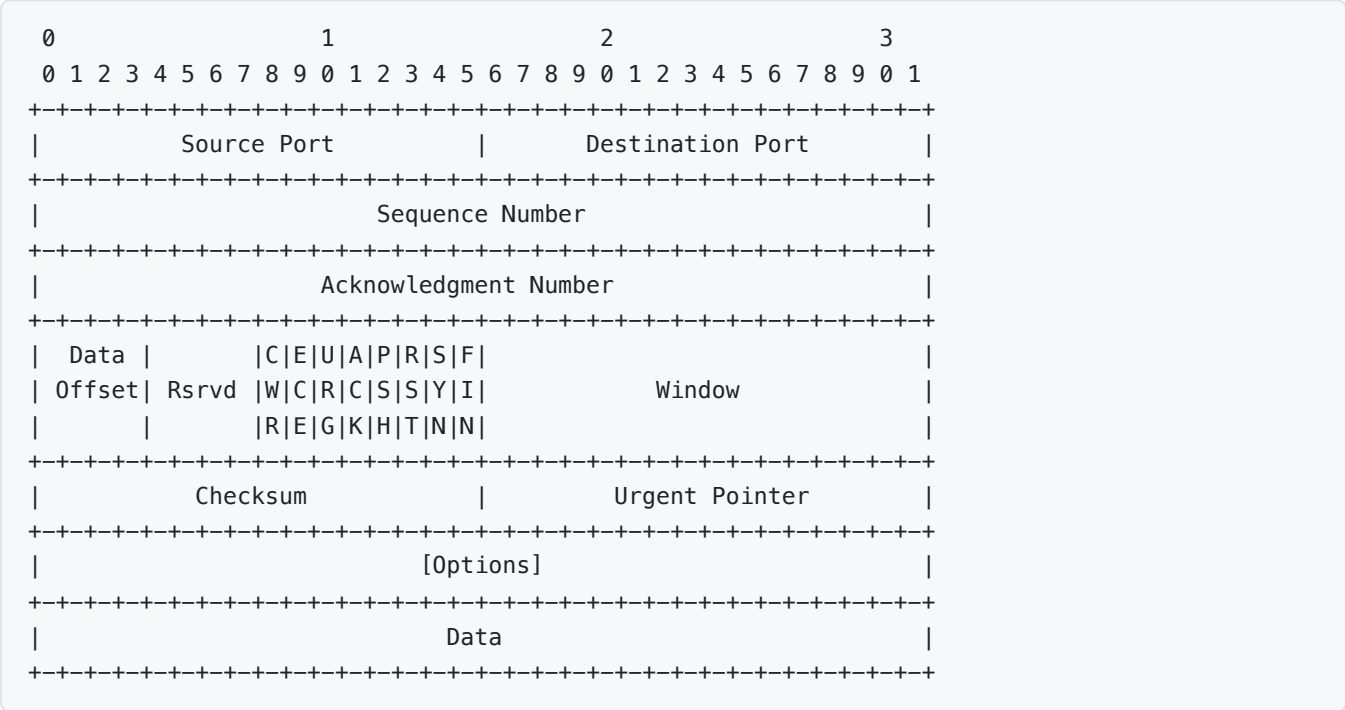
第 7 章 TCP 报文段：字段级解剖与三种序号

7.1 为什么把 TCP 头拆到比特级

TCP 头只有 20 字节（不含选项），但它是**全 Internet 最被反复解析的 20 个字节**。理解它的每一个比特，才能回答这些实际问题：

- 为什么窗口字段只有 16 位，却能支持 1 GB 的窗口？（窗口缩放选项）
- 为什么序号要从随机数开始？（防止旧连接的数据被误认）
- 为什么有 `PSH` 这个几乎没人用的标志位？
- 为什么 `URG`/紧急指针被认为是设计失误？

7.2 TCP 头逐字段表（依据 RFC 9293 §3.1 的图 1）



字段	位宽	含义	关键细节与坑
Source Port	16	源端口	与源 IP 一起构成“哪条连接”的一半
Destination Port	16	目的端口	服务器侧用它做**复用分派**（80/443/22…）
Sequence Number	32	本段第一个数据字节的序号	**SYN 置位时它就是 ISN**，第一个数据字节是 ISN+1
Acknowledgment Number	32	期望收到的下一个序号	只有 ACK 置位时有效；建立连接后**每个段都带**
Data Offset	4	头长度，单位 4 字节（“32 位字”）	最小 5（20 B），最大 15（60 B）→ **选项最多 40 B**
Reserved	3（+1）	保留	历史上被中间设备乱用（如“实验位”），是 middlebox 干扰的重灾区

字段	位宽	含义	关键细节与坑
CWR / ECE	1 / 1	拥塞窗口减小 / ECN 回显	ECN 三步：ECT 标记 → 路由器 CE → 接收方 ECE 回显
URG	1	紧急指针有效	几乎无人实现，**公认的设计失误**
ACK	1	确认号有效	除首个 SYN 外几乎总是 1
PSH	1	提示接收方尽快交付应用	现代实现基本忽略（由缓冲区策略决定）
RST	1	重置连接	用于“这个连接不存在/出错了”
SYN	1	建立连接，同步序号	**占 1 个序号**
FIN	1	发送方没有更多数据	**占 1 个序号**
Window	16	接收窗口（以字节为单位，未缩放）	最大 65535；要更大必须用窗口缩放选项
Checksum	16	伪首部 + TCP 头 + 数据的反码和	见第 6 章
Urgent Pointer	16	紧急数据偏移	与 URG 一起被弃用
Options	0–40 B	MSS、窗口缩放、SACK、时间戳…	见 7.4 节

> **一个必须纠正的直觉**：`Data Offset` 的单位是**4 字节**，不是字节。写 lab 或解析抓包时若按字节算，会把 20 读成 5 或 80。

7.3 三种序号：seqno / absolute seqno / stream index

这是 CS144 check2 的核心，也是全课最容易混的一组概念。check2 用一个只有三个字母的字符串 `cat` 把它讲清楚了：

元素	syn	c	a	t	fin
seqno（32 位，会回绕，起点是随机 ISN）	$2^{32}-2$	$2^{32}-1$	0	1	2
absolute seqno（64 位，不回绕，含 SYN/FIN）	0	1	2	3	4
stream index（64 位，不回绕，**不含** SYN/FIN）	—	0	1	2	—

三种表示的定义（check2 原文的表格复述）：

属性	Sequence Numbers	Absolute Sequence Numbers	Stream Indices
起点	ISN（随机）	0	0
是否含 SYN/FIN	含	含	**不含**

属性	Sequence Numbers	Absolute Sequence Numbers	Stream Indices
位宽	32 位，会回绕	64 位，不回绕	64 位，不回绕
别称	seqno	absolute seqno	stream index

换算关系：

```
absolute_seqno = stream_index + 1               (SYN 占了 absolute 0)
seqno          = (absolute_seqno + ISN) mod 2^32
absolute_seqno = unwrap(seqno, ISN, checkpoint) ← 需要"就近"假设
```

为什么“就近”是必须的：给定 seqno = 17、ISN = 0，absolute 可能是 17、$2^{32}+17$、$2^{33}+17$… 无穷多个。TCP 用“**离 checkpoint 最近的那个**”来消歧，而 checkpoint 在实现里就是“第一个未装配的序号”。

```
// check2 要求实现的三个函数（语义，不是完整实现）
Wrap32 Wrap32::wrap( uint64_t n, Wrap32 zero_point );        // absolute → seqno（一行）
uint64_t Wrap32::unwrap( Wrap32 zero_point, uint64_t checkpoint ) const;  // seqno → absolute (
```

`unwrap` 的干净写法（**推断，非 handout 原文**，仅供理解）：

```
candidate = (checkpoint & ~0xFFFFFFFF) | (seqno - zero_point).value
若 candidate + 2^31 < checkpoint → candidate += 2^32
若 candidate > checkpoint + 2^31 → candidate -= 2^32
返回 candidate
```

核心思想：先把高位“对齐”到 checkpoint 附近，再根据“差超过半个序号空间就翻转”修正。

7.4 关键选项（options）

选项	长度	作用	为什么重要
MSS（Maximum Segment Size）	4 B	通告“我能收的最大段”	只在 SYN 里出现；以太网典型 1460 B（1500 − 20 IP − 20 TCP）
Window Scale	4 B	窗口左移位数（0–14）	16 位窗口 × 2^{14} ≈ **1 GB**；只在 SYN/SYN-ACK 协商，**双向各自独立**
SACK-Permitted + SACK	2 B + 变长	选择性确认	见 5.5.3；一个段最多塞 3–4 个块
Timestamps	10 B	发送时间戳 + 回显	RTT 测量（绕开 Karn）、**PAWS**（防序号回绕）
NOP / EOL	1 B	对齐填充	选项必须按 4 字节对齐

> **窗口缩放的陷阱**：缩放因子只能在握手时协商，且**双方各用一个**。抓包时若只看到 `Win=65535` 别急着说“窗口小”——Wireshark 会显示 `calculated window size`，那才是缩放后的值。

7.5 为什么 ISN 要随机

Week 3 的 lecture notes 用“**connection incarnation（连接的转世）**”来解释：

1. 同一对 `(IP, port)` 可能被复用给新连接；
2. 如果序号总是从 0 开始，**旧连接的迟到段**会被新连接误认为合法数据（会造成数据错乱，甚至安全漏洞）；
3. 所以新连接选一个**难以猜测**的随机 ISN；接收方看到 `SYN=1` 且序号“看起来不属于当前连接”时，就知道这是新的一世。

配套机制：

- **PAWS（Protection Against Wrapped Sequences）**：用时间戳判断“这个段是不是太老了”，比 32 位序号本身更可靠；
- **TIME_WAIT 等 2×MSL**：让旧连接的残余段在网络中自然死亡（第 8.4 节）；
- **ISN 随机化还要防攻击**：可预测的 ISN 会导致 TCP 序列号预测攻击（注入伪造段），这也是现代内核用加密随机数生成 ISN 的原因。

7.6 常见误解

误解	事实
“序号是字节计数，从 0 开始”	序号是**32 位回绕**的，起点是**随机 ISN**，且 SYN/FIN 各占一个。
“一个段就是一个字节”	一个段可携带最多 MSS 字节；段的序号是**第一个字节**的序号。
“window 字段就是我能收多少”	它是**未缩放**值；实际窗口 = 该值 × 2^{scale}。而且它是“从 ackno 起还能收多少”，不是“缓冲区总大小”。
“SYN 和 FIN 不占序号”	都占 1 个。这就是为什么“流长度 N 字节”占用序号空间 N+1（+1 for SYN）或 N+2（+SYN+FIN）。
“选项可有可无，解析可以跳过”	选项长度由 `Data Offset` 决定；解析错会导致把选项当数据，是抓包分析最常见的低级错误。

第 8 章 TCP 连接管理：三次握手、四次挥手、状态机与 TIME_WAIT

8.1 从“可靠字节流”到“连接”：为什么需要握手

Week 3 的 lecture notes 把这个推导做得极漂亮，值得完整复述：

问题：TCP 要提供双向字节流，但双方一开始**都不知道对方的初始序号**，也不知道对方是否愿意通信。

如果只发一个 `SYN`：接收方无法确认“发送方收到了我的序号”，也无法确认“发送方还活着”。

于是需要三步（notes 里的推导顺序）：

```
① 我：{seqno=12340, SYN=1, data="abcdef", FIN=1},  {ACK=0, ackno=空, window=1000}
                                                          ↑ 我还不知道对方序号，所以不能带 ackno
② 对方：{seqno=9999, SYN=1, data="", FIN=1}, {ACK=1, ackno=12348, window=1000}
        ↑ 既确认了我的 SYN（ackno=12340+1+7=12348），又通告了自己的 ISN
③ 我：{ACK=1, ackno=10001, window=1000}
        ↑ 确认对方的 SYN（9999+1+1(FIN)=10001），握手完成
```

三个段的组合就是**三次握手（three-way handshake）**：

段	标志	作用	消耗序号
SYN	SYN=1, ACK=0	客户端通告 ISN	1（SYN）
SYN-ACK	SYN=1, ACK=1	服务端通告 ISN + 确认客户端	1（SYN）
ACK	ACK=1	确认服务端 SYN	0

为什么不是两次：两次无法让**服务端**确认“客户端收到了我的 SYN”。服务端会一直保留半开连接（SYN_RCVD），直到超时——这正是 SYN flood 攻击的原理。

为什么不是四次：第二段的 ACK 可以和 SYN 合并（piggyback），因为服务端此时已经知道自己的 ISN。

8.2 同时打开（simultaneous open）：不是“经典三次握手”但仍合法

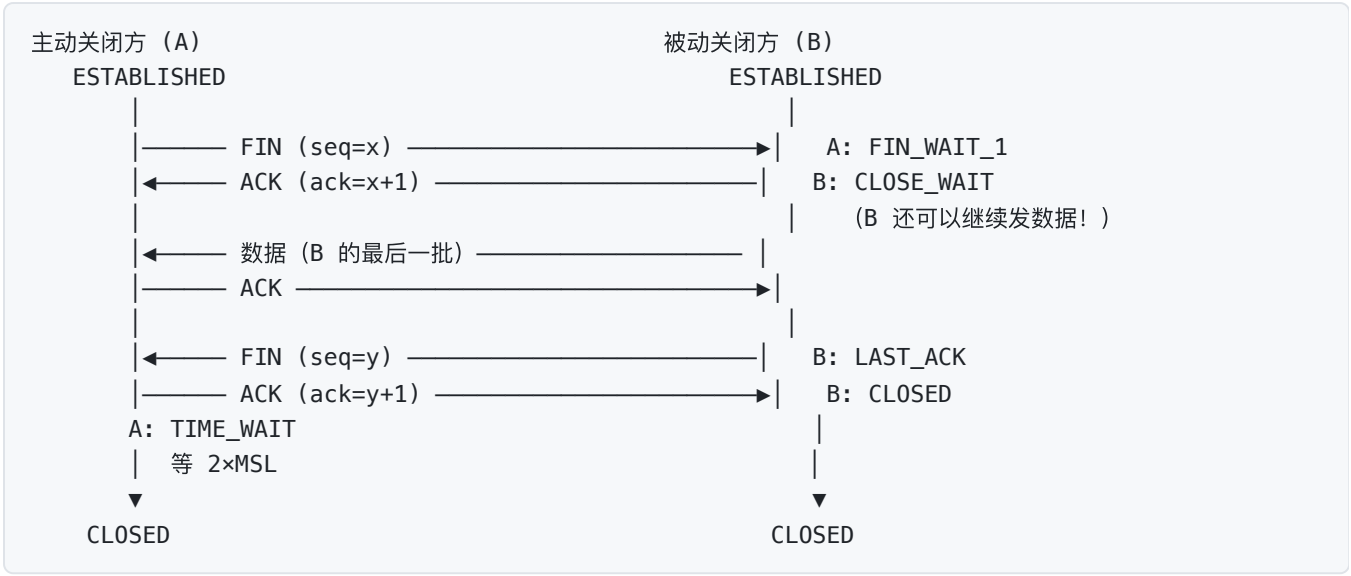
notes 里特意问了一个问题：如果双方**同时**发 SYN 会怎样？

```
A: SYN(ISN_A) ──────────────► B
A: ◄────────────── SYN(ISN_B) B
A: SYN+ACK(ack=ISN_B+1) ────► B
A: ◄──── SYN+ACK(ack=ISN_A+1) B
```

结果是**四次**报文，但状态机仍然收敛到 ESTABLISHED。这说明：**“三次握手”是常见路径，不是协议硬性要求**——TCP 状态机允许各种交错。考试里若出现“同时打开”，按状态图走（CLOSED → SYN_SENT → SYN_RCVD → ESTABLISHED）。

8.3 四次挥手：为什么关闭要四步

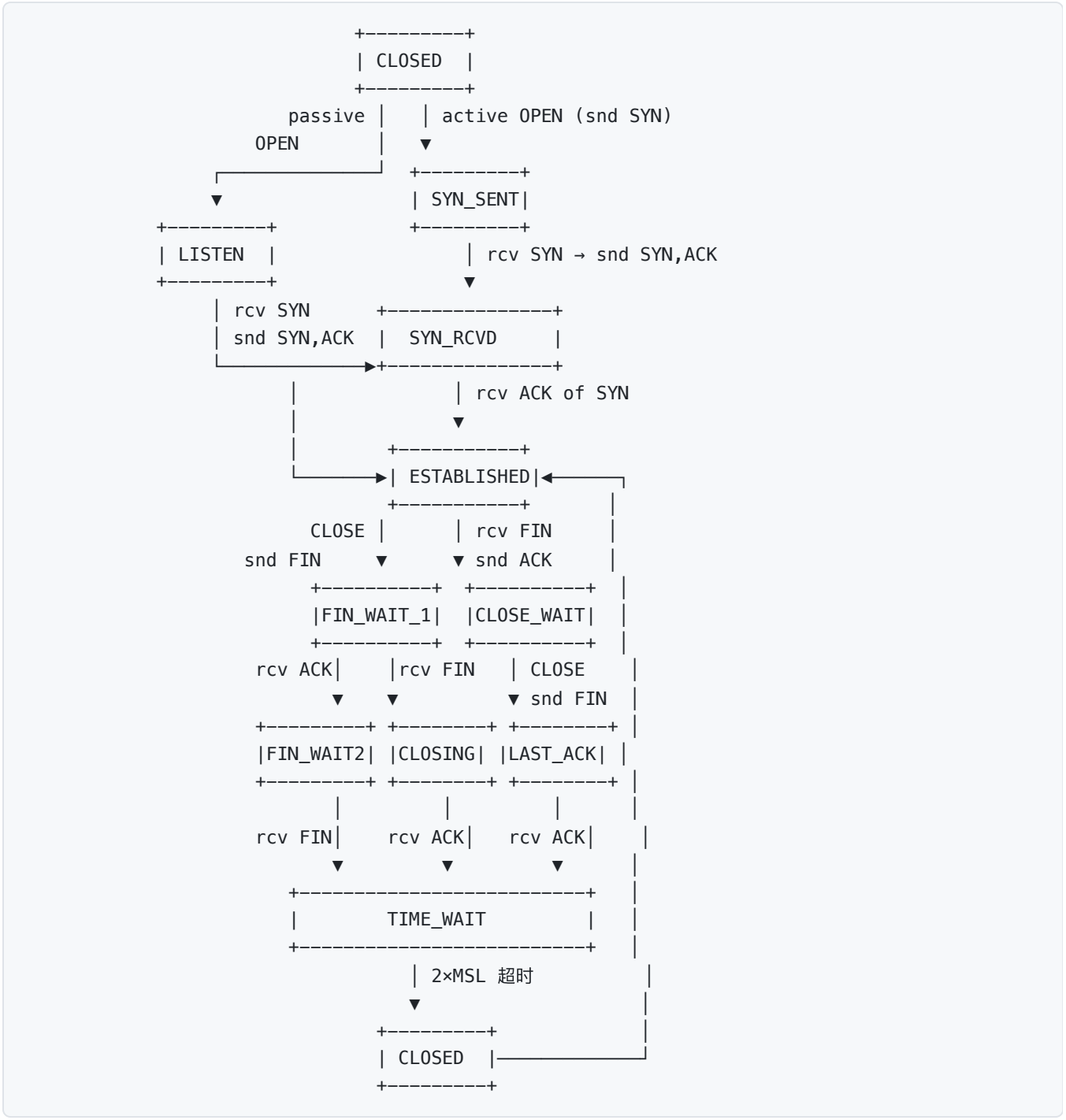
关闭是**两个方向独立**的，所以需要两对 FIN/ACK：



为什么不能把 FIN 和 ACK 合并（多数情况下）：B 收到 FIN 时，应用可能还有数据要发（半关闭，half-close）。只有“B 也已经没有数据要发”时，FIN 才能搭在 ACK 上，退化成三步。

Week 3 的 lab 里你也能量到这个行为：`tcp_native` 里按 `ctrl-D` 只关闭**一个方向**，另一端仍可继续发数据。

8.4 状态机（RFC 9293 图 5 的精简版）



必须记住的 11 个状态：`CLOSED`、`LISTEN`、`SYN_SENT`、`SYN_RCVD`、`ESTABLISHED`、`FIN_WAIT_1`、`FIN_WAIT_2`、`CLOSE_WAIT`、`CLOSING`、`LAST_ACK`、`TIME_WAIT`。

两个最常被问的状态：

状态	谁在这里	意味着什么	危险信号
`CLOSE_WAIT`	**被动**关闭方	收到对方的 FIN 并回了 ACK，但**自己的应用还没 close()**	大量 CLOSE_WAIT = **应用忘了 close**（代码 bug）
`TIME_WAIT`	**主动**关闭方	已回最后一个 ACK，等 2×MSL	大量 TIME_WAIT = 短连接太多（可用连接池/长连接缓解）

> 运维口诀：**CLOSE_WAIT 多查代码，TIME_WAIT 多查架构。**

8.5 TIME_WAIT 为什么必须存在

两个理由（RFC 9293 §3.5 与 Week 3 notes 的“两个将军”讨论）：

1. **保证最后一个 ACK 能到达对方**。如果这个 ACK 丢了，对方会重传 FIN；如果 A 已经 CLOSED，收到重传的 FIN 会回 RST，对方就会认为“连接异常终止”。TIME_WAIT 让 A 还能**重发 ACK**。
2. **让旧连接的残余段在网络中消亡**。等待 2×MSL 后，可以确信旧连接的所有段都已消失，新连接（同四元组）不会收到“上一世”的数据。

MSL（Maximum Segment Lifetime）：RFC 793 建议 2 分钟，实际 Linux 取 60 s（`tcp_fin_timeout`），于是 TIME_WAIT 约 60 s。

工程上的四个缓解手段：

手段	机制	风险
`SO_REUSEADDR`	允许绑定处于 TIME_WAIT 的端口	相对安全，服务器常用
`tcp_tw_reuse`	允许**主动**发起新连接时复用 TIME_WAIT 的端口（需时间戳）	需 PAWS 保护，NAT 环境下可能出错
长连接 / 连接池	从源头减少短连接	需处理 idle 超时与半开连接
让**客户端**主动关闭	把 TIME_WAIT 放在客户端（端口多、无所谓）	需协议配合（如 HTTP 的 `Connection: close` 语义）

不要做的事：`tcp_tw_recycle`（已在内核 4.12 移除）——它依赖“同一源 IP 的时间戳单调递增”，在 NAT 后面会随机丢连接。

8.6 RST：不体面的结束方式

RST 用于三类场景：

1. **连接不存在**：收到一个不属于任何 socket 的段（如服务器重启后收到旧连接的包）→ 回 RST；
2. **异常中止**：应用调用 `SO_LINGER` 且 linger=0 时 `close()`，或进程崩溃后内核清理；
3. **半开连接检测**：一方已消失，另一方发数据收到 RST。

```
收到 RST 后：
  - 立刻丢弃所有在途数据与缓冲
  - 应用侧 read() 返回 ECONNRESET
  - 不发送任何 ACK（RST 不需要确认）
```

安全注意：RST 可被伪造（只要四元组与序号在窗口内），是“连接重置攻击”的基础；QUIC 用加密的连接 ID + 认证的帧来防这类攻击。

8.7 常见误解

误解	事实
“三次握手是为了同步两个序号”	不止。它同时确认**双向可达**与**双方序号**，并防止旧连接的 SYN 造成伪连接。
“四次挥手可以合并成三次”	只有在“被动方也没有数据要发”时才能合并；否则 FIN 必须等应用 close。
“TIME_WAIT 是浪费，应该关掉”	它保护的是**可靠性**与**连接隔离**；关掉会引入“旧数据污染新连接”的诡异 bug。
“CLOSE_WAIT 是内核问题”	几乎总是**应用没 close**。内核已经做完它该做的（回 ACK），在等应用。
“服务端不应该有 TIME_WAIT”	谁先 `close()` 谁进 TIME_WAIT。HTTP/1.1 里通常是服务端先关（`Connection: close`），所以服务端 TIME_WAIT 多很正常。

第 9 章 TCP 实现细节：Nagle、延迟 ACK、糊涂窗口与零窗口

这一章的内容在课本里常被塞进“实现细节”一节草草带过，但**真实网络的性能问题一大半出在这里**。Week 4 的《TCP Particulars》专门讲这些“不性感但要命”的机制。

9.1 MSS 与 MTU：为什么是 1460

概念	含义	典型值
MTU（Maximum Transmission Unit）	**链路层**能承载的最大帧载荷	以太网 1500 B；IPv6 最低要求 1280 B
MSS（Maximum Segment Size）	**TCP** 能承载的最大数据量	以太网 1460 B = 1500 − 20(IP) − 20(TCP)

```
以太网帧 = [14B 头][20B IP][20B TCP][最多 1460B 数据][4B FCS]
                              └──── MSS ────┘
                    └──────── MTU = 1500 ────────┘
```

路径 MTU 发现（PMTUD）：中间链路 MTU 更小时，路由器回 ICMP “Fragmentation Needed”，发送方据此调小 MSS。**坑**：很多防火墙把这类 ICMP 丢了 → 出现“小包能通、大包卡死”的经典故障（MTU 黑洞）。隧道（VPN、PPPoE）会额外吃掉 8–60 字节，所以隧道场景常把 MSS clamp 到 1360 左右。

9.2 Nagle 算法（RFC 896）：小包合并

问题：Telnet 式应用每敲一个键就发一个字节 → 40 字节的头带 1 字节数据，**开销 4000%**。

Nagle 规则：

> 只要还有**未被确认的小段**（小于 MSS），就把新数据**缓冲**起来，直到攒满一个 MSS **或** 收到前一个的 ACK 再发。

```
无 Nagle:   [1B][1B][1B][1B]  →  4 个 41 字节的包
有 Nagle:   [1B] 等 ACK → [3B] 一起发  →  2 个包
```

Nagle + 延迟 ACK 的致命组合（经典 40 ms 延迟）：

```
客户端                                服务端
  |── 发 "a"（小包）──────────────►|
  |                                  |  收下，但"延迟 ACK"：等第二个段或 500ms
  |◄──────── 不发 ACK ──────────────|
  |  Nagle：有未确认小段 → 憋住 "b"  |
  |                                  |
  |          …… 卡住 40ms ……         |
  |◄──────── 延迟 ACK 到达 ──────────|
  |── 发 "ab" ──────────────────────►|
```

解法：`TCP_NODELAY`（禁用 Nagle）。游戏、SSH、RPC、Redis 客户端几乎都开它。代价是小包变多——所以现代实现（如 Linux 的 `TCP_NODELAY` + `TCP_CORK`）提供了更细的控制。

9.3 延迟 ACK（delayed ACK，RFC 1122）

规则：

1. 收到数据后**不立刻**回 ACK，等一小段（典型 40–200 ms，上限 500 ms）；
2. 若在这段时间内收到**第二个**满长段，立刻 ACK（一次确认两个段）；
3. 收到**乱序**段 → **立刻** ACK（让发送方尽快快速重传）；
4. 收到**重复**段 → 立刻 ACK。

收益：ACK 数量减半，且给“搭车（piggyback）”创造机会。 **代价**：与 Nagle 组合会把时延放大到几十毫秒；对 RTT 敏感的应用（游戏）必须关掉其中之一。

> **lab 相关**：check3 的 `TCPSender` 不实现延迟 ACK（它是发送方）；但你在做 check3 的 hands-on（与 Linux TCP 对话）时会**真实遇到**这个行为——你会发现小数据的往返明显变慢。

9.4 糊涂窗口综合征（Silly Window Syndrome, SWS）

场景：接收方缓冲区只空出 1 字节，就通告 window=1；发送方于是发 1 字节数据 + 40 字节头。**双方都很忙，但吞吐接近 0**。

```
接收方缓冲: [■■■■■■■■■■■■□]  ← 只剩 1 字节空间
   → 通告 window=1
发送方:      → 发 1 字节
接收方:      → 应用读走 1 字节 → 又通告 window=1
   …… 无限循环，每个 RTT 只传 1 字节 ……
```

两侧的对策：

侧	对策	规则
接收方（Clark 算法）	不要通告小窗口	直到“可用空间 ≥ min(MSS, 缓冲区一半)”才通告非零窗口；否则通告 0
发送方（SWS 避免）	不要发小段	满足以下之一才发：① 攒满 MSS；② 所有排队数据 + 无未确认小段（Nagle）；③ 收到 ACK 且窗口 ≥ 半个缓冲区

零窗口（zero window）：接收方缓冲区满 → 通告 `window=0`。发送方必须**停止发送**，并启动 **persist timer（坚持定时器）** 周期性发**窗口探测（window probe）**：一个字节的“试探”段，逼接收方回一个带当前窗口的 ACK。

> **check3 的对应规则**：当接收方通告 `window=0` 时，`push()` **假装窗口为 1**，允许发出 1 个字节。handout 特意强调：这是**唯一**为“零窗口”开的特例，`TCPSender` **不能真的把窗口记成 1**；而且“窗口满了”与“窗口为零”是两件事。

9.5 Keepalive：一条有争议的机制

- **默认关闭**（Linux 需 `SO_KEEPALIVE` 打开），默认空闲 2 小时才探测。
- 作用：检测**半开连接**（对端崩溃/断电/中间 NAT 超时）。
- 争议：① 2 小时太长，对多数应用无意义；② 会替应用“决定”连接死亡；③ 在计费网络里会产生额外流量。
- **实践**：应用层心跳（HTTP/2 PING、WebSocket ping、gRPC keepalive）比 TCP keepalive 更可控。

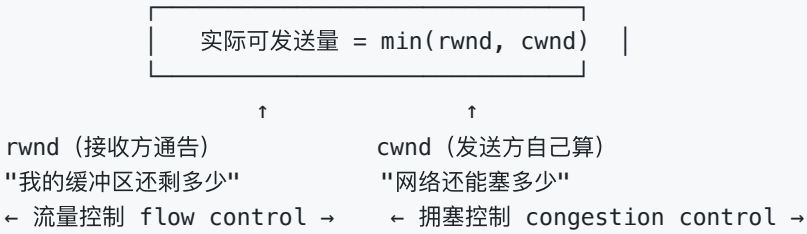
9.6 常见误解

误解	事实
“Nagle 会降低吞吐”	它降低的是**小包开销**；对大批量传输几乎无影响（数据本来就满 MSS）。它伤的是**交互式小包时延**。
“延迟 ACK 最多延迟 500 ms”	500 ms 是**上限**；Linux 实际用 40 ms 起始并动态调整（`TCP_ATO_MIN`）。
“window=0 就是连接挂了”	只是暂时不收。发送方应持续探测；连接是否存活由超时/keepalive 判断。
“关了 Nagle 就一定更快”	在“大量小写 + 高 RTT”场景，关 Nagle 会让包数暴涨，可能因拥塞控制反而更慢。要测量。
“MSS 由发送方决定”	MSS 是**接收方通告**的（“我能收多大”），发送方取双方较小值。

第 10 章 流量控制与拥塞控制的分界

10.1 两个窗口，一个上限

Week 5 讲义的第一句话就把问题摆清楚：TCP 提供“**流控的（flow-controlled）**双向字节流”——发送方尊重**接收方**的容量。但“接收方的容量”不等于“网络的容量”。



维度	流量控制	拥塞控制
保护对象	**接收方**的内存	**网络**（瓶颈链路与路由器队列）
信号来源	接收方的 `window` 字段	丢包、RTT 上升、ECN 标记
失控后果	接收方缓冲区溢出（丢数据）	拥塞崩溃（吞吐崩塌）
决策者	接收方（被动通告）	发送方（主动探测）
时间尺度	每段	每 RTT

10.2 自时钟（self-clocking）：为什么用“窗口”而不是“速率”

Week 5 讲义的问答非常关键：

> **问**：为什么不直接限制“发送速率”，而要限制“窗口”？ **答**：自时钟很强大！如果速率差 1% 会怎样？

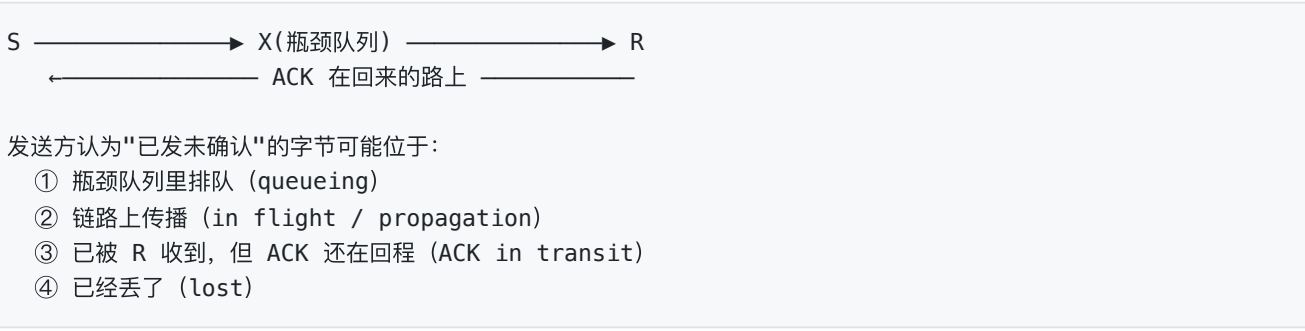
直觉：窗口是“**在途字节数**”的上限，而 ACK 的到达节奏**天然等于瓶颈链路的服务节奏**。发送方每收到一个 ACK 就发一个新段，于是发送节奏自动与网络同步——不需要测量带宽、不需要定时器。

```
速率控制（开环）：   按 5 Mb/s 定时发 → 若实际瓶颈是 4.9 Mb/s，队列每秒涨 0.1 Mb/s → 最终丢包
窗口控制（闭环）：   保持 W 字节在途 → 瓶颈快就 ACK 快、发得快；瓶颈慢就 ACK 慢、发得慢
                          ↑ 自动跟踪，误差不累积
```

唯一的例外：窗口为 0 或应用没有数据时，自时钟会“停摆”，所以需要 persist timer 与 pacing（BBR 用 pacing 重建节奏）。

10.3 发送方眼里，包可能在哪

Week 5 讲义给了一个“单流单跳”模型：发送方 S —— 路由器 X —— 接收方 R，链路 1 Gb/s，传播 1 s。



这四种状态**发送方无法区分**——这就是为什么拥塞控制只能用“统计信号”（丢包、RTT 变化）去猜。

10.4 窗口太小或太大分别会怎样

窗口	结果	公式/现象
极小（如 1 字节）	吞吐被窗口限制	吞吐 $\leq W/RTT$（1 字节 / 100 ms = 80 bit/s）
恰好 BDP	理想：链路满、队列空	$W = R \times RTT$
远大于 BDP	队列堆积 → 排队时延暴涨 → 丢包	多余部分 = $W - BDP$ 字节堆在瓶颈
无限大	**拥塞崩溃**	大量重传，网络做无用功

吞吐的硬上限：吞吐 $= \min(\text{链路速率}, W/RTT)$。这条公式解释了两件事：为什么长肥管道（LFN, Long Fat Network）需要窗口缩放（第 7.4 节），为什么“加带宽”不一定提速（窗口不够）。

10.5 拥塞崩溃（congestion collapse）与公平性

Week 5 讲义给了 1980 年代的真实教训：接收方通告大窗口 → 发送方猛发 → 路由器大量丢包 → 重传又加重负担 → **网络吞吐崩塌**。notes 里的判据很精炼：

> 有用功应随需求增长而增长。导数小于 1 可以接受，但**导数不能为负**。

讲义还给了一张“三个流如何分 30 单位带宽”的表（原文数据），这张表值得背：

A→a	B→b	C→c	总计	评价
20	10	0	30	最大利用率（但 C 饿死）
10	0	10	20	对 C 最好（但 B 饿死）
0	0	20	10	**崩溃**（总利用反而最低）

A→a	B→b	C→c	总计	评价
15	5	5	25	**最大最小公平（max-min fair）**：最差的那个尽可能好
16	6	4	26	**比例公平（proportionally fair）**

α-fairness 的统一框架（notes 提到“资源分配的数学”）：

$$\max \sum_i U(x_i), \qquad U(x) = \begin{cases} \log x & \alpha = 1 \text{（比例公平）} \\ \frac{x^{1-\alpha}}{1-\alpha} & \alpha \neq 1 \\ -x & \alpha \to \infty \text{（最大最小公平）} \end{cases}$$

- $\alpha \to 0$：最大化总吞吐（不管公平）；
- $\alpha = 1$：比例公平（“提升 x 需要伤害别人超过 1/x”）；
- $\alpha \to \infty$：最大最小公平（最差的最好）。

为什么这跟工程有关：不同的 TCP 变种隐含不同的效用函数。Reno 大致追求“比例公平”，而 BBR 追求“最大吞吐 + 低排队”，两者共存时会出现**公平性冲突**（BBR 抢占 Reno 的带宽）——这是学术界争论多年的问题。

10.6 拥塞信号：三种可选输入

信号	例子	优点	缺点
丢包	超时、3 dupACK	明确、易实现	已经太晚（队列已溢出）；误判无线误码
时延	RTT 上升、排队时延	早于丢包（预警）	受路由变化/ACK 压缩干扰；测量噪声大
显式标记	ECN（ECE/CWR）、AQM（RED/CoDel）	不需要丢包就能通知	需要路由器配合，部署缓慢

Week 5 讲义的最后一页就是这三行——**“Ideas for congestion signals: RTT 上升、丢包”**，而 ECN 是第三条路（第 12.4 节）。

10.7 常见误解

误解	事实
“流控就是拥塞控制”	流控保护接收方内存，拥塞控制保护网络。rwnd 与 cwnd 是两个独立变量，取最小值。
“窗口大就快”	窗口只决定**上限**；实际速率还受 RTT 与丢包率约束（第 11.5 节的公式）。
“自时钟永远有效”	应用没数据、窗口为 0、ACK 被延迟聚合时都会失效，需要 pacing/persist 兜底。
“公平性是自然属性”	不是。AIMD 有收敛到公平的性质，但 RTT 不同的流、UDP 流、BBR 流都会破坏公平。

第 11 章 拥塞控制（一）：AIMD、慢启动与 Reno

11.1 AIMD：一句话算法

Week 5 周三讲义《How congestion control?》给出的方案，原文几乎是伪代码：

> - 从很小的 cwnd 开始（例如 3 个段）
> - **成功时**（段被完全确认）：每个 RTT 增加 1 个段；等价地，**每收到一个字节的 ACK：`cwnd += MSS/cwnd`**
> - **丢包时**：假设是拥塞，**cwnd 减半**
> - 丢包的推断依据：① 某段发出很久仍未确认（超时）；② 多个后发的段已被确认（重复 ACK）

这就是 **AIMD（Additive Increase, Multiplicative Decrease）：加性增、乘性减**。

```
// 拥塞避免阶段的逐 ACK 更新（等价于每 RTT +1 MSS）
cwnd += MSS * MSS / cwnd;         // 收到一个 MSS 的确认
// 或者按字节：cwnd += MSS / cwnd * acked_bytes
```

手算校验：若 cwnd = 10 MSS，一个 RTT 内会收到 10 个 ACK，每个 ACK 加 `MSS/10`，合计 `MSS` → 每 RTT 恰好 +1 MSS。✅

11.2 锯齿图：AIMD 的稳态



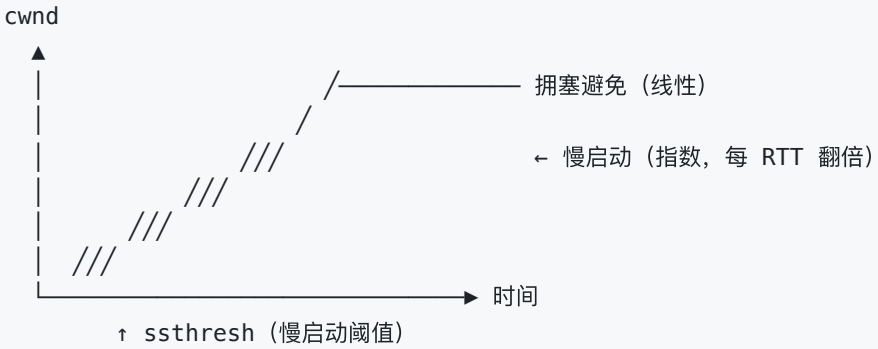
为什么是“加性增”：多个流共享瓶颈时，线性增长能让它们**逐步收敛到公平**（11.4 节）。 **为什么是“乘性减”**：讲义自己问了一句“**为什么丢包要砍这么狠？**”——答案是：丢包意味着队列已满，必须**快速**把在途量降到安全区，否则持续丢包会让吞吐崩塌；而“减半”是 O(1) 的粗粒度操作，实现简单且收敛快。

11.3 慢启动（slow start）：名字骗人，其实是指数增长

问题：从 3 个段开始、每 RTT +1，要爬到 1000 个段需要 1000 个 RTT（跨洋链路 ≈ 100 s）。太慢。 **但也不能**一上来就用接收方窗口那么大（那会直接拥塞崩溃）。

折中：慢启动

阶段	触发	增长方式	退出条件
慢启动	连接开始 / 超时后	**每个 ACK 加 1 MSS**（每 RTT 翻倍）	`cwnd >= ssthresh`
拥塞避免	慢启动结束	每 RTT 加 1 MSS	丢包



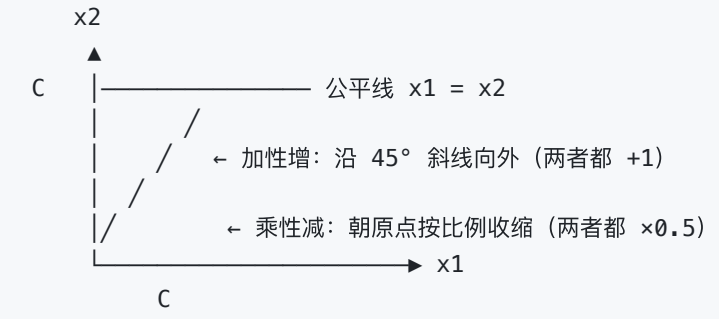
初始窗口（IW）的历史（版本差异，务必要注意）：

规范	IW
早期实现	1 MSS
RFC 5681	$\mathbf{min}(4 \cdot MSS,\ \mathbf{max}(2 \cdot MSS,\ 4380\ \mathrm{B}))$
RFC 6928（实验性，已被广泛采用）	$\mathbf{min}(10 \cdot MSS,\ \mathbf{max}(2 \cdot MSS,\ 14600\ \mathrm{B}))$
CS144 讲义（简化）	**3 个段**

> 讲义用 3 个段是为了教学与 lab 简单；**真实 Linux 默认 IW=10**。考试若问“初始窗口是多少”，先看题目给的是哪套假设。

11.4 为什么 AIMD 能收敛到公平（一个漂亮的小证明）

设两个流共享带宽 C，当前速率 (x_1, x_2)，总需求 $x_1 + x_2 > C$（过载）。



- **乘性减**把点沿“过原点的射线”拉回效率线附近（保持两流的**比例**）；
- **加性增**沿 45° 线向外推（增加**绝对量**相同）；

- 反复迭代，点会**螺旋收敛到公平线**。若反过来（AI 换成 MI、MD 换成 AD），就会收敛到**不公平**。

这是“为什么偏偏是 AIMD”最本质的回答：**乘性减保持比例、加性增抹平差异**。

11.5 AIMD 的吞吐公式（Mathis 公式）

对稳态 AIMD，丢包率为 p 时：

$$\text{吞吐} \approx \frac{MSS}{RTT} \cdot \frac{1}{\sqrt{p}} \cdot C$$

（C 为常数，常见写法 $\frac{1.22\,MSS}{RTT\sqrt{p}}$。）

三个直接推论：

1. **吞吐 ∝ 1/RTT**：RTT 大的流吃亏 → 这是“RTT 不公平”问题的根源；
2. **吞吐 ∝ 1/√p**：丢包率从 1% 升到 4%，吞吐只减半（不是减到 1/4）；
3. **要跑满 10 Gb/s、RTT 50 ms、p=1e-5**：需要 cwnd ≈ 数千个 MSS —— 这就是数据中心要调大窗口、用 DCTCP 的原因。

11.6 Tahoe、Reno、NewReno

版本	丢包检测	丢包后动作	关键改进
Tahoe	超时 / 3 dupACK	`ssthresh=cwnd/2`，`cwnd=1`，重进慢启动	基线
Reno	3 dupACK	**快速重传 + 快速恢复**（`cwnd=ssthresh+3`，之后回 `ssthresh`）	不必等超时，也不必回到 1
NewReno（RFC 6582）	一个窗口内多个丢包	用“部分确认（partial ACK）”逐个修复，每修一个洞 `cwnd` 减 1	多洞场景不再反复进出快速恢复
SACK 类（RFC 6675）	精确知道缺哪些段	每 RTT 填多个洞	多洞场景效率最高

```
Tahoe:   丢包 → cwnd=1          → 一个 RTT 只发 1 个段，恢复极慢
Reno:    丢包 → cwnd=ssthresh → 直接以半速继续
NewReno: 丢包 → 逐个洞修复      → 多个丢包也能在一个恢复期内搞定
```

11.7 超时 vs 快速重传的窗口处理对照

事件	ssthresh	cwnd	阶段
3 个重复 ACK	$\max(\text{FlightSize}/2,\ 2\,MSS)$	$ssthresh + 3\,MSS$（并随 dupACK 膨胀）	快速恢复

事件	ssthresh	cwnd	阶段
快速恢复中收到新 ACK	不变	$ssthresh$	拥塞避免
超时	$\max(\text{FlightSize}/2,\ 2\,MSS)$	$1\,MSS$	慢启动

记忆：超时 = 网络出了大问题（可能连续丢包/路径断了），所以要“回到起点重新摸索”；**dupACK = 局部丢包**，半速即可。

11.8 常见误解

误解	事实
“慢启动很慢”	它是指数增长，每 RTT 翻倍。名字来自“相对于直接跳到接收方窗口而言很慢”。
“丢包就一定要减半”	那是**拥塞**假设下的策略。无线误码、reordering 造成的丢包不该减半——这催生了 BBR、CUBIC 的改进思路。
“cwnd 是字节数”	它是**字节数**（历史上按 MSS 计），但很多教材写“以段为单位”。写代码时统一单位是最大的 bug 来源之一。
“AIMD 保证公平”	只在“所有流 RTT 相同、都遵守 AIMD”时成立。RTT 差异、UDP、并行连接都会破坏它。
“ssthresh 一旦设定就不再变”	每次丢包事件都会重设 `ssthresh = max(FlightSize/2, 2MSS)`。

第 12 章 拥塞控制（二）：CUBIC、BBR 与新一代算法

12.1 为什么 Reno 不够用

Reno 的三个结构性缺陷，在现代网络（高 BDP、无线、数据中心）里被放大：

缺陷	机制	后果
高 BDP 下恢复太慢	每 RTT 只 +1 MSS	10 Gb/s × 100 ms 的管道要跑满需 cwnd≈83 000 MSS，线性增长要几十分钟
RTT 不公平	吞吐 ∝ 1/RTT	近距离流挤死远距离流
把丢包等同于拥塞	无线误码 → 减半	无线链路上吞吐剧烈抖动

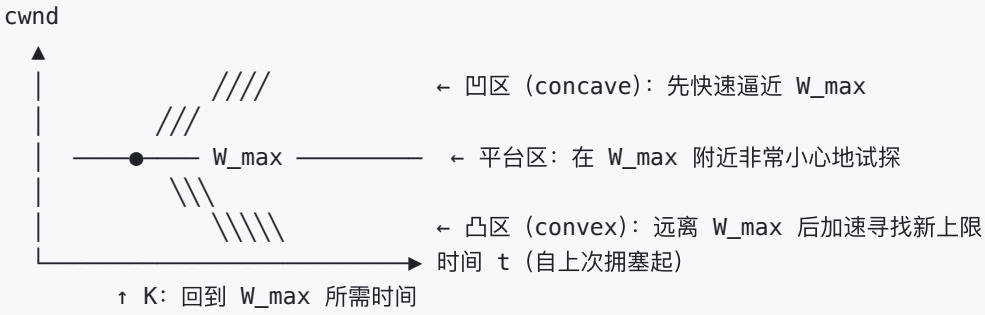
三个改进方向：**改增长函数**（CUBIC）、**改信号来源**（BBR 用带宽/时延）、**让路由器提前告知**（ECN/AQM）。

12.2 CUBIC（RFC 9438）：用三次函数替代线性

核心思想：cwnd 不再是时间的线性函数，而是**以“上次丢包时的窗口 W_{max}”为参照的三次函数**，并且**只与时间有关，与 RTT 无关**（解决了 RTT 不公平）。

$$W_{cubic}(t) = C\,(t-K)^3 + W_{max}, \qquad K = \sqrt[3]{\frac{W_{max}(1-\beta)}{C}}$$

其中 $C = 0.4$（RFC 9438 的默认缩放常数），$\beta = 0.7$（丢包后保留 70% 窗口，而不是 Reno 的 50%）。



四个要点：

1. **凹区**：刚丢包后不要急着回到 W_{max}（那里刚拥塞过），先快速接近再放缓——比 Reno 的线性更“聪明”。
2. **平台区**：在 W_{max} 附近增长极慢（导数近 0），减少“刚探测到上限又立刻丢包”的振荡。
3. **凸区**：如果网络容量增长了（W_{max} 已过时），则加速向外探测。
4. **TCP-friendly 区域**：CUBIC 仍保证在“小窗口/RTT 大”时不比 Reno 差（RFC 9438 有专门的 TCP-friendly 区域约束）。

现状：Linux 自 2.6.19 起默认 CUBIC；Windows 10/11、macOS 也已支持。它是今天 Internet 上**占比最高**的拥塞控制算法。

12.3 BBR：不用丢包，用“带宽 × 时延”建模

BBR（Bottleneck Bandwidth and Round-trip propagation time） 由 Google 提出（BBRv1 论文 2016，ACM Queue；后续 BBRv2/v3 为 IETF ICCRG 草案，**不是 RFC**）。

核心洞察：一条路径有两个物理量：

- **BtlBw（瓶颈带宽）**：这条路径最快能送多少（由最慢的那条链路决定）；
- **RTprop（最小往返传播时延）**：路径的“空载”RTT。

理想的操作点是：**在途数据 = BtlBw × RTprop（即 BDP），队列为空**。这正是第 10.4 节说的“理想窗口”，BBR 试图**直接测量这两个量**，而不是靠丢包去猜。

```
测量方式（滑动窗口最大值/最小值）：
   BtlBw  = 一段时间内观测到的最大"交付速率"（delivery rate）
   RTprop = 一段时间内观测到的最小 RTT（min RTT）

状态机（简化）：
   STARTUP ──(带宽3轮不涨)──▶ DRAIN ──▶ PROBE_BW ⇄ PROBE_RTT
                                          │          │
                                   周期性地把 pacing gain 在
                                   1.25 / 0.75 / 1 / 1 / ... 之间循环
                                   以探测"还有没有更多带宽"
```

对比项	Reno/CUBIC	BBR
主要信号	丢包	交付速率 + min RTT
队列	主动填满到丢包为止（bufferbloat）	试图保持空队列
无线/随机丢包	误判为拥塞，降速	不因随机丢包降速
公平性	AIMD 收敛	与 CUBIC 共存时**可能抢占**（v1 的著名问题）
部署	内核默认	Google 内部/部分 CDN、Linux 可加载模块
深队列表现	时延高	时延低，但可能“打爆”浅队列

BBRv1 被批评的核心问题：它倾向于**多占带宽**（因为它的探测周期总是“多要一点”），在多流共存时对 CUBIC 不公平；且依赖 pacing 与精确的交付速率估计，在 ACK 聚合（ACK compression）下会估错。BBRv2 引入丢包响应与 ECN 支持来缓解。

12.4 ECN 与 AQM：让路由器“提前说话”

问题：丢包是“事后”信号——等它发生，队列已经满了，时延已经飙了。

ECN（Explicit Congestion Notification，RFC 3168）：IP 头里的两个比特（ECT(0)/ECT(1)）由发送方设置；路由器发现队列过长时，**不丢包，而是把 CE（Congestion Experienced）置位**；接收方收到 CE 后，在 TCP 头里用 **ECE** 回显；发送方看到 ECE 就**像丢包一样减窗**（并用 CWR 确认）。

```
发送方 ──[ECT]──▶ 路由器（队列将满）──[CE]──▶ 接收方
                                                │
   发送方 ◀──────────[ECE]──────────────────────┘
     ↓ 减窗（半速），并置 CWR
```

AQM（Active Queue Management）：路由器主动管理队列，而不是“满了才丢”。

算法	策略	特点
Tail drop（默认）	满了丢新来的	简单，但造成全局同步（global synchronization）
RED（Random Early Detection）	队列超过阈值就**随机**丢/标记	提前预警，但参数难调
CoDel	按“包在队列里待了多久”丢包	与速率无关，抗 bufferbloat，Linux 默认可选

算法	策略	特点
PIE / FQ-CoDel	组合流隔离 + 时延控制	家用路由器常见

Bufferbloat：家用路由器/电缆调制解调器缓冲区过大 + tail drop → 队列长期满 → RTT 从 10 ms 涨到 1000 ms。CoDel/FQ-CoDel 就是为此设计的。

12.5 一张总表：五代拥塞控制

代	代表	信号	增长	减窗	解决的问题
1	Tahoe	丢包	慢启动 + 线性	到 1	崩溃
2	Reno/NewReno	丢包	慢启动 + 线性	减半	快速恢复
3	SACK 类	丢包 + 精确缺口	同上	减半	多洞恢复
4	CUBIC	丢包	三次函数	×0.7	高 BDP、RTT 公平
5	BBR / DCTCP / QUIC 内置	速率/时延/ECN	模型驱动	模型更新	排队时延、无线误码

12.6 常见误解

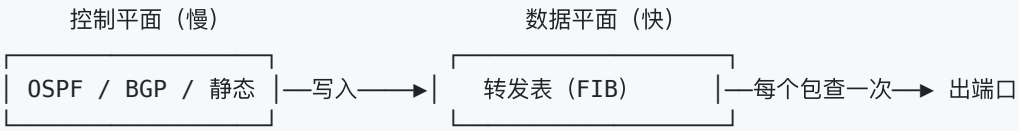
误解	事实
“CUBIC 是 RFC 但 BBR 也是标准”	CUBIC 是 RFC 9438；BBR 只有 IETF 草案与论文，**不是 RFC**。
“BBR 不发包就不会丢包”	BBR 仍会丢（浅队列 + 突发），只是它不把丢包当主要信号。
“ECN 需要端到端全支持”	需要**两端 + 沿途路由器**都支持；任一处不支持就退化为丢包。
“AQM 只影响时延”	它也影响公平性与吞吐（RED 的全局同步问题、CoDel 的流隔离缺失）。
“拥塞控制只是 TCP 的事”	QUIC 在用户态实现自己的拥塞控制（可插拔），数据中心有 DCTCP/TIMELY/Swift 等专用算法。

第 13 章 路由与转发表：最长前缀匹配

13.1 转发（forwarding）与路由（routing）的区别

这是网络里最容易混的一对词：

	转发 Forwarding	路由 Routing
时间尺度	每个包（纳秒级）	秒到分钟级
做什么	查表 → 决定从哪个口出去	算出表的内容
在哪做	数据平面（硬件/快速路径）	控制平面（软件/协议）
类比	邮局分拣员看邮编	邮政总局规划邮路



13.2 前缀（prefix）与 CIDR

讲义《Internet Routing》从最基础的定义讲起：

> 目的地被**聚合**成路由前缀。`18.31.0.82` 其实是 32 位串 `00010010 00111110 00000000 01010010`。

写法	含义	地址范围
`18.0.0.0/8`	前 8 位固定	2^{24} 个地址（`00010010*`）
`18.31.0.0/17`	前 17 位固定	2^{15} 个地址
`18.31.0.0/24`	前 24 位固定	256 个地址（典型家庭/机房子网）
`0.0.0.0/0`	默认路由	全部（“其它都从这儿走”）

子网可以递归划分：`/8` 切成若干 `/16`，`/16` 再切成 `/24`……这就是 CIDR（Classless Inter-Domain Routing）的核心——**用掩码长度代替“类别”**。

13.3 最长前缀匹配（Longest Prefix Match, LPM）

规则：在**所有匹配**的表项里，选**前缀最长**的那条。

```
转发表：
   0.0.0.0/0        → eth0（默认路由）
   18.0.0.0/8       → eth1
   18.31.0.0/16     → eth2
   18.31.7.0/24     → eth3

目的地址 18.31.7.9 匹配了全部 4 条 → 选 /24 → eth3   ✅ 最长前缀
目的地址 18.31.8.9 匹配前 3 条        → 选 /16 → eth2
目的地址 18.99.1.1 匹配前 2 条        → 选 /8  → eth1
目的地址 9.9.9.9   只匹配默认           → eth0
```

为什么必须最长前缀：它同时实现了“**默认路由 + 例外**”和“**聚合 + 细化**”。没有 LPM，BGP 就没法既通告大块前缀又保留例外。

13.4 如何快速做 LPM：三种数据结构

结构	查找复杂度	内存	更新	适用
线性表	O(N)	小	快	N 很小（教学、lab）
二叉前缀树（binary trie）	O(W)=O(32)	大（每层一个节点）	中	软件路由
多比特 Trie / 压缩 Trie（Patricia）	O(W/k)	中	中	高速软件转发
TCAM（三态内容寻址存储器）	**O(1)**（一次并行比较）	极贵、功耗高	慢	硬件交换机
哈希 + 多级	近似 O(1)	中	快	现代可编程交换机（如 P4 的 LPM 表）

```
二叉 Trie 示意（只画前 3 位）：
                root
             0 /    \ 1
              /      \
           (0)        (1)
          /   \      /   \
       (00)  (01)  (10)  (11)
         \             /
       [18/8]      [18.31/16] ...
   查找：沿目的地址的比特逐位下降，记录"沿途最后一个有路由的节点"→ 那就是最长前缀
```

lab 相关（推断）：如果课程后续的 IP 路由器 checkpoint 要求你做 LPM，最稳的起点是**按前缀长度分桶的哈希表**（32 个桶，从 /32 往下找第一个命中）——实现 30 行、性能足够；想更快再上压缩 Trie。

13.5 转发表里还有什么

除了“前缀 → 出端口”，真实表项还带：

字段	作用
Next hop（下一跳 IP）	以太网里还需要 ARP 解析成 MAC
Metric / cost	多路径选优（OSPF cost、BGP 属性）
Admin distance	多协议优先级（直连 > 静态 > OSPF > BGP）
出接口	`eth0`、`wg0`、`tun144`

字段	作用
标志	是否黑洞、是否 ECMP 成员

ECMP（Equal-Cost Multi-Path）：多条等价路径同时用，按五元组哈希分流。它是数据中心“胖树”拓扑的基础，也带来“同一条流走同一路径”的约束（否则会乱序）。

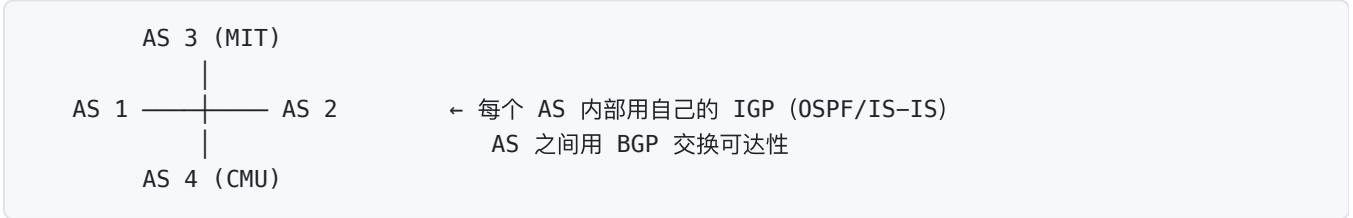
13.6 常见误解

误解	事实
“路由表按目的地址精确匹配”	是**前缀匹配 + 最长优先**，不是精确匹配。
“默认路由是 0.0.0.0”	是 `0.0.0.0/0`（长度为 0 的前缀），含义是“匹配一切”。
“前缀一定是 8/16/24”	CIDR 允许任意长度（/1 到 /32）。`/31` 用于点对点链路（RFC 3021），`/32` 是主机路由。
“转发要查路由协议”	转发只查 FIB；协议在控制平面跑，二者解耦。
“LPM 必须用 TCAM”	软件实现（trie、分桶哈希）在 10–100 Gb/s 量级完全够用；TCAM 是功耗换速度。

第 14 章 路由协议：OSPF、BGP 与 Internet 的经济学

14.1 自治系统（AS）：把 Internet 切成“政策域”

讲义定义：**AS = 一组有相同策略的网络，跑单一路由协议，通常由单一机构管理，有唯一 ASN**（早期 16 位，现在 32 位）。



讲义给的 ASN 例子（原文）：MIT = 3，CMU = 9，AT&T = 7018/6341/5074…，UUNET = 701/702/284…，Sprint = 1239/1240/6211…

两类协议：

类别	全称	作用域	代表	目标
IGP	Interior Gateway Protocol	AS 内部	OSPF、IS-IS、EIGRP、RIP	**最短路径**（性能）
EGP	Exterior Gateway Protocol	AS 之间	**BGP**	**政策**（钱与策略）

14.2 OSPF：链路状态 + Dijkstra

工作方式：

1. 每个路由器发现自己的直连链路与代价（cost，常按带宽反比）；
2. 用 **LSA（Link State Advertisement）** 把“我连了谁、代价多少”**泛洪**到整个区域；
3. 每个路由器拼出**完整的拓扑图**，跑 **Dijkstra** 算最短路径树；
4. 得到“到每个目的地的最佳下一跳”，写入 FIB。

```
OSPF 泛洪 → 每台路由器都有同一张"地图" → 各自算最短路 → 表一致
    优点：收敛快、无环、支持 ECMP
    代价：地图要一致（区域划分来限制规模）；泛洪占带宽
```

区域（area）：把大 AS 切成多个 area（骨干 area 0），减少泛洪范围与 LSDB 规模。

14.3 BGP：路径向量（path vector）与政策

BGP 的关键设计：它不通告“距离”，而是**通告整条 AS 路径**。

```
距离向量：   "到 d 的距离是 3"          ← 邻居无法判断路径是否经过自己（环）
路径向量：   "到 d 的路径是 (2,1)"      ← 邻居看到自己就在路径里 → 直接拒绝（防环）
             "到 d 的路径是 (1)"   ← 通告给邻居 2
```

讲义的原话：**Path-Vector = 距离向量的扩展，支持灵活的路由政策；关键思想是通告整条路径。**

14.3.1 导入与导出政策（这是 BGP 的灵魂）

政策	问题	常见答案
Import policy	从邻居学到的路由怎么处理?	选最优（按属性排序）
Export policy	向邻居通告哪些路由?	**取决于商业关系**

三条铁律（讲义“Implementing Transit / Peering”的结论）：

通告给谁	可以通告哪些路由	理由
客户（customer）	从 provider、peer、customer 学到的全部 + 自己的	客户付钱，要“全表”
提供商（provider）	只通告从**客户**学到的 + 自己的	提供商的线路是买来的，不能白送别人
对等体（peer）	只通告从**客户**学到的 + 自己的	对等是“免费互通”，不能把转接当免费

```
                provider
        (付钱)  /        \  (免费, settlement-free peering)
               /          \
         本 AS ———————— peer
               \  (收钱)
                customer
```

为什么这样设计：如果 AS 把“从 provider 学到的路由”通告给 peer，peer 就能免费使用它的上游——**转接（transit）被白嫖**。所以政策不是技术限制，而是**商业模型的直接编码**。

14.3.2 BGP 的属性与选路顺序

顺序	属性	方向	说明
1	**LOCAL_PREF**	越高越优	**本 AS 内部**政策，控制**出方向**流量；不跨 AS 传递
2	**AS_PATH 长度**	越短越优	注意：**短 ≠ 快**（AS 路径短不代表时延低）
3	**MED**（Multi-Exit Discriminator）	越低越优	告诉**邻居**“从哪个入口进我更好”，控制**入方向**
4	eBGP 优于 iBGP	—	外部学到的优先
5	到 next-hop 的 IGP cost 最小	—	**hot-potato routing**：尽快把包甩出本 AS
6	Router ID	任意	只为止平局

Hot-potato（热土豆）路由：谁都不想替别人多扛流量，所以“离出口最近”优先。副作用是**流量会在 AS 边界来回横跳**，且 IGP 权重的微小变化会引起大范围流量迁移（讲义专门列了这个问题）。

14.3.3 iBGP 的两种拓扑

方式	结构	优点	缺点
Full mesh	每个 eBGP 路由器与所有其它路由器建 iBGP 会话	简单、收敛快	$O(n^2)$ 会话，不可扩展
Route reflection	每台与“反射器”建会话，反射器再扩散	可扩展	需要设计层次、防环

14.4 BGP 的脆弱性：没有认证的全球协议

讲义用两页讲 **BGP 劫持（hijack）**，核心一句：

> **设计缺陷：没有认证，也没有校验机制。**

```
正常:   AS 18 拥有 18.0.0.0/16，向邻居通告
劫持:   某个 AS 也通告 18.0.0.0/16（或更长的 /24），
         由于"最长前缀优先 + 缺少来源验证"，流量被吸走
```

讲义给的案例：2018 年 **Amazon Route 53 前缀被劫持**，攻击者伪造 DNS 服务器，盗走约 **1700 万美元**的以太币（MyEtherWallet 事件）。学术界常用的另一个案例是 2008 年巴基斯坦电信“屏蔽 YouTube”导致全球 YouTube 短暂不可达。

缓解措施：

机制	作用	部署状态
RPKI / ROA	用密码学签名声明“这个前缀属于这个 AS”	逐步部署，覆盖率仍不完整
BGPsec	对路径做逐跳签名	部署极少（性能与政策阻力）
前缀过滤（prefix filter）	邻居手工/自动限制可通告的前缀	普遍但依赖运维
最大前缀数限制	防止邻居误配置泄露全表	普遍
RPKI 的 maxLength	限制更具体前缀的长度	关键（否则 /24 劫持仍可行）

14.5 Internet 的“商业地理”：从 1995 到今天的扁平化

讲义用三张图讲拓扑演化，这条线非常值得记住：

时期	结构	特征
1995 前	单一政府骨干 + 区域网	层级清晰，无商业问题
1995–2005	多商业骨干互联（BGP 出现）	层级：国家骨干 → 区域 ISP → 本地接入 → 用户
今天	**扁平化 + CDN 下沉**	内容提供商自建骨干、CDN 深入 ISP 内部

讲义给的数据：2009 年约 150 家公司贡献一半流量；2014 年降到 30 家；**2022 年 6 家公司（Facebook、Amazon、Google、Apple、Netflix、Microsoft）占 65% 的流量**。

含义：今天的“路由”很大程度上是**内容提供商与 ISP 之间的直连与对等**，而不是传统的层级转接。这也解释了为什么 BGP 政策问题（如 2021 年 Facebook 因 BGP 撤销导致全球宕机）影响如此之大。

14.6 常见误解

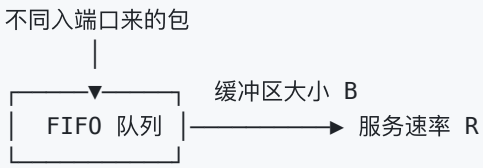
误解	事实
“BGP 选最短 AS 路径 = 最快路径”	不是。BGP 先看 LOCAL_PREF（政策），而且 AS 路径长度与物理距离/时延无关。
“OSPF 和 BGP 是竞争关系”	分工：OSPF 管 AS 内，BGP 管 AS 间。BGP 的 next-hop 可达性还要靠 IGP 提供。
“BGP 有认证”	原始 BGP 没有来源认证；TCP MD5（RFC 2385）只保护**会话**，不保护**路由声明**。RPKI 才管声明。
“Internet 有一个中心路由表”	没有。每个 AS 有自己的视图，默认路由 + 政策导致**不对称路由**是常态。

误解	事实
“AS 号就是 IP 段”	AS 是**政策域**，一个 AS 可以拥有多个不相邻的前缀，一个前缀也可能由多个 AS 通告（anycast）。

第 15 章 交换与排队：FIFO、优先级、WFQ 与令牌桶

15.1 默认排队：FIFO（先来先服务）

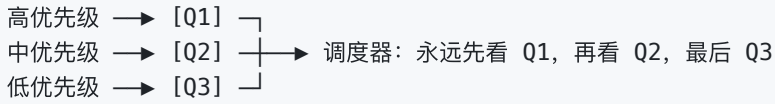
讲义《Packet Switching》的结论很直接：**交换机与路由器默认用 FIFO（即 FCFS）队列**——没有优先级、没有速率保证，先到的先走。



FIFO 的问题：一个突发流可以把队列占满，让所有流一起遭殃（**bufferbloat** 与**队头阻塞**）。

15.2 严格优先级（strict priority）

规则：只有当所有更高优先级队列为空时，才服务低优先级队列。



优点	缺点
高优先级流“看到”一个没有低优先级流量的网络（时延最小）	高优先级流可以**永久饿死**低优先级流
实现简单（几个队列 + 一个判优器）	必须能控制高优先级流量的大小，否则低优先级无意义

讲义的建议：**限制高优先级流量的比例**；若需要“按比例”，改用 WFQ。

15.3 公平队列（FQ）与加权公平队列（WFQ）

15.3.1 从“按包轮转”到“逐比特轮转”

- 按包轮转：每个流轮流发一个包 → **包大小不同则数据速率不同**（大包流占便宜）。

- 逐比特轮转（bit-by-bit）：每个流轮流发一个比特 → 真正等分带宽，但**那就不是分组交换了**（无法切开包）。

WFQ 的妙招：按“逐比特轮转下的完成时间”给包排序发送——既保持分组交换（不切包），又近似逐比特公平。

```
三个流的包（数字 = 逐比特方案下的完成时刻）：
  F1: [3] [2] [1]      F2: [5] [2]      F3: [4] [3] [1]
  发送顺序（按完成时刻升序）：1, 1, 2, 2, 3, 3, 4, 5
  ⇒ 每个流拿到的速率趋于相等
```

定理（讲义引用 Demers/Keshav/Shenker 1990）：FQ 下每个包的离开时间，比“逐比特方案”晚不超过 L_{max}/R（L_{max} 为最大包长，R 为链路速率）。且流数增多时仍成立。

15.3.2 加权版本

给每个流一个权重 ϕ_i，$\sum_i \phi_i = 1$，则**流 i 至少获得 $\phi_i R$ 的速率**。这就是 WFQ（Weighted Fair Queueing）。

```
φ1=3/4 ⟶ 获得 3/4 R
φ2=1/4 ⟶ 获得 1/4 R
实现：把"逐比特完成时间"按 1/φ 缩放后再排序
```

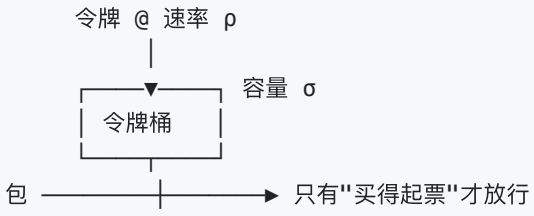
> **现实中**：几乎所有商用设备都实现 WFQ 变种（Linux 的 `tc` 有 `fq_codel`、`hfsc`、`cbq`；家用路由器也在用）。但**公网不提供端到端时延保证**——讲义明确说：那需要全路径路由器协同，实际靠“过度部署（over-provisioning）+ 优先级”解决。

15.4 令牌桶（token bucket / leaky bucket）：限制突发

问题：即使 WFQ 保证了平均速率，一个流仍可能**瞬间**发一大串包（burst），把队列打爆。

令牌桶规则：

- 令牌以速率 ρ 注入，桶容量 σ；
- 每个包必须“买票”（消耗等于包长的令牌）才能发；
- 桶满则令牌丢弃（不允许攒无限信用）。



性质：任意长度 t 的时间窗内，发出的比特数 $\le \sigma + \rho t$——这就是 **(σ, ρ) 约束**（也称 leaky bucket regulator）。

级联定理：若到达是 (σ, ρ) 约束的，队列服务速率 $\phi R > \rho$，缓冲 $b > \sigma$，则**离开也是 (σ, ρ) 约束的** ⇒ 下一跳的到达仍然受约束。

端到端时延上界（讲义给出的结果）：若每跳 $b_i > \sigma$ 且 $\phi_i R > \rho$，

$$\tau \le \sum_{i=1}^{4} \left(\frac{p}{r_i} + \frac{l_i}{c} \right) + \frac{3\sigma}{\rho}$$

讲义例题（原文数据）：要给一个流 10 Mb/s、1000 字节包、端到端时延 < 4.7 ms，路径 10 km/1 Gb/s + 100 km/100 Mb/s + 10 km/1 Gb/s，$c = 2 \times 10^8$ m/s。

- 固定部分：$120\,\text{km}/c + 8000\,\text{bit} \times (1/10^9 + 1/10^8 + 1/10^9) = 0.7$ ms；
- 留给队列 4 ms，两跳各 2 ms；
- 每跳队列 $\le 2\,\text{ms} \times 10\,\text{Mb/s} = 20000$ bit = 2500 B；
- 故 $\rho = 10$ Mb/s，$\sigma \le 20000$ bit，WFQ 每跳 $\phi_i R \ge 10$ Mb/s，缓冲 ⩾ 2500 B。

15.5 常见误解

误解	事实
“WFQ 能保证带宽就一定不丢包”	保证的是**平均速率**；突发仍可能超过缓冲而丢包，需要令牌桶整形。
“优先级越高越好”	高优先级过多会让低优先级饿死，最终整网只有高优先级流在跑。
“队列越长越安全”	长队列 = 大缓冲 = 高时延（bufferbloat）。AQM 的思路是“宁可早丢，不要久等”。
“逐比特轮转可以实现”	不能切包；WFQ 是它的**包级近似**，误差有 L_{max}/R 上界。

第 16 章 NAT、家庭网络与 P2P 穿透

16.1 “The eras tour of home networking”：九个层级

Week 6/7 的讲义用“时代之旅”的方式，把家庭网络从 1970 年代串到今天。这条演化线极好地解释了**为什么会有 NAT**。

Level	形态	新增部件	关键问题
0	终端 + 串口	UART、tty	一台机器独占一条线
0.1	终端 + 调制解调器	modem	用电话线延伸串口（两端“以为”直连）
1	家庭 PC 上网	SLIP/PPP	在字节流上**切出数据报边界**（SLIP 加长度）
2	电缆调制解调器	以太网	modem 改说以太网
3	多台 PC	**交换机**	交换机维护“MAC → 端口”表，**不看 IP**
4	家庭子网	**路由器**	ISP 只需一条 `/24` 路由，而不是每台主机一条 `/32`

Level	形态	新增部件	关键问题
5	无线	Wi-Fi AP	ISP 无法给每个设备分配公网 IP
6	代理（proxy/SOCKS5）	应用层中继	每台 PC 都要配置代理，麻烦
7	透明代理	中间盒	PC 不知道代理存在
8	NAPT	地址+端口翻译	只改 IP/端口，不重建字节流

Level 3 的痛点（讲义的推导）：没有家庭路由器时，ISP 的路由表里要写

```
18.1.2.7/32 → eth1     ← PC1
18.1.2.8/32 → eth1     ← PC2
```

每加一台设备就多一条，**不可扩展**。Level 4 引入家庭路由器后：

```
ISP 路由表:    18.1.2.0/24 → eth1 via 18.1.2.2   ← 一条搞定
家庭路由表:    0.0.0.0/0     → eth1 via 18.1.0.1  ← 默认路由指向 ISP
               18.1.2.0/24   → eth0               ← 本地子网
```

这就是**路由聚合（aggregation）**的威力，也是 CIDR 存在的理由。

16.2 NAPT 的规则表

NAPT（Network Address and Port Translation）：把“私有 IP:端口”映射到“公网 IP:端口”，并维护一张**转换表**。

```
内网                                    公网
10.0.0.7:6101  ─┐
                ├─→ 18.17.4.19:2012 ──→ 151.101.1.69:443 (stackoverflow)
10.0.0.8:6101  ─┘          ↑
                  同一个公网 IP，靠"端口"区分不同内网主机
```

内网侧	公网侧	目的
10.0.0.7:6101	18.17.4.19:2012	151.101.1.69:443
10.0.0.8:6101	18.17.4.19:2013	151.101.1.69:443

规则的生命周期：新建连接时插入；连接关闭或**一段时间未使用**时被垃圾回收（GC）。这就带来一个经典问题：**NAT 的超时比 TCP 的 keepalive 短** → 长连接被悄悄掐断（“NAT 超时”故障）。

容量上限：一台内网主机到某个公网 IP 的并发连接数 ≤ 端口数（65536）——这就是为什么“一台机器扫全网”会被 NAT 卡住。

16.3 四种 NAT 映射类型（RFC 3489 的分类）

类型	规则形式	谁能从外面打进来
Full cone	任意源 → 固定公网端口 → 固定内网主机	任何人（只要知道公网 IP:端口）
Restricted cone	只允许“内网主机曾发过包的目的 IP”	该 IP 的任意端口
Port restricted cone	只允许“曾发过包的目的 IP:端口”	该 IP 的该端口
Symmetric	每个“目的 IP:端口”分配**不同**的公网端口	几乎不可能（端口不可预测）

> 这是 P2P 打洞（NAT traversal）的核心分类。**Symmetric NAT 是打洞的噩梦**，因为它对每个目的地换端口，导致对方无法预测该往哪打。

16.4 P2P 穿透的四种方案

方案	机制	代价
公共服务器中转（Level 9a）	文件放公网服务器	服务器带宽/存储
TURN 中继（Level 9b）	双方都连中继，中继转发	中继带宽（贵）
端口映射/UPnP（Level 9c）	手工在 NAT 上加规则	需权限、不安全
NAT 打洞 + STUN + 会合服务器（Level 9d）	① 用 STUN 问“我的公网地址是什么” ② 用**会合服务器（rendezvous）**交换地址 ③ 双方同时向对方公网地址发包，打穿锥形 NAT	对 Symmetric NAT 无效

```
内网 A ──► STUN 服务器（无状态）："我从外面看是谁？" → 得到 18.241.9.7:2222
内网 A ──► 会合服务器：把我的公网地址告诉 B
内网 B ──► 会合服务器：拿到 A 的地址
A ──直接发包──► B 的公网地址（同时 B 也向 A 发）→ 两边的 NAT 都建立了映射 → 打通
```

现代 WebRTC 把这套流程标准化为 **ICE**（Interactive Connectivity Establishment）：先试直连，再试 STUN 打洞，最后回退 TURN。

16.5 NAT 破坏了什么

被破坏的性质	后果
端到端可达性	外网无法主动连内网（需要端口映射/打洞）
无状态性	NAT 要维护连接状态，重启即断所有连接

被破坏的性质	后果
分层的“透明性”	NAT 改 IP/端口 → 校验和要重算、应用层协议（FTP/SIP）要 ALG 特殊处理
IP 作为身份	多个主机共享一个公网 IP → 基于 IP 的限流/封禁会误伤

IPv6 的解法：地址足够多，每家每户一个 `/64`，不需要 NAT——但防火墙仍然需要（“有状态防火墙”替代了 NAT 的部分功能）。

16.6 常见误解

误解	事实
“NAT 是安全机制”	它只是**副作用**上提供了“外网进不来”。它不是防火墙，也不检查内容。
“NAT 只改 IP”	NAPT 还改**端口**，并且必须重算 TCP/UDP 校验和。
“打洞总能成功”	Symmetric NAT + 对称地址池会让打洞失败，必须回退中继（TURN）。
“IPv6 就不需要 NAT”	不需要地址转换，但仍需要**有状态防火墙**来阻止非预期入站。

第 17 章 应用层：HTTP/1.1、HTTP/2、HTTP/3、DNS 与 QUIC

17.1 应用层要的四种“可靠性语义”

Week 2 的 notes 已经把应用需求分成四类（2.5 节），这里把它和协议对应起来：

需求	语义	协议	为什么不用 TCP
取一小段数据	幂等查询	DNS（UDP 53）、DHCP	一次往返就够，TCP 握手太贵
可靠字节流	有序、不丢	HTTP/1.1、SMTP、IMAP（TCP）	—
低时延实时流	宁可丢不可等	RTP/WebRTC（UDP）	重传对实时流是毒药
多路复用 + 加密	多流独立	HTTP/2（TCP）、HTTP/3（QUIC）	TCP 的队头阻塞

17.2 HTTP/1.1：文本协议与它的三个瓶颈

```
GET /hello HTTP/1.1\r\n
Host: cs144.keithw.org\r\n
Connection: close\r\n
\r\n
```

这四行就是 check0 里你手工敲进 `telnet` 的内容——**HTTP 是纯文本、行式、以空行结束头部**。三个瓶颈：

瓶颈	机制	影响
队头阻塞（HOL）	一个 TCP 连接上请求必须按序响应	前一个响应慢，后面全等
每连接一次握手	每开一个连接要 TCP 三次握手 + TLS 握手	高 RTT 下开销巨大
头部冗余	每个请求都带完整 Cookie/UA	小请求也可能几百字节头

缓解手段：**持久连接（keep-alive）**、**流水线（pipelining，实际很少用）**、**多域名分片（domain sharding）**、**雪碧图/合并资源**。这些都是“绕过协议缺陷”的工程 trick。

17.3 HTTP/2：二进制分帧 + 多路复用

特性	做法	收益
二进制分帧	把请求/响应切成 FRAME（HEADERS/DATA/SETTINGS...）	解析快、可扩展
多路复用	一个 TCP 连接上并发多个 stream（stream id）	消除 HTTP 层 HOL
HPACK 头压缩	静态表 + 动态表 + 哈夫曼	头从几百字节降到几十
服务器推送	服务端主动推资源	实践效果不佳，HTTP/3 已弱化

遗留问题：HTTP/2 的多路复用跑在**一条 TCP** 上，而 TCP 是**按序交付**的——只要有一个 TCP 段丢失，**所有 stream 一起等**（TCP 层 HOL）。这就是“HTTP/2 在丢包率高的链路上反而比 HTTP/1.1 慢”的著名现象。

17.4 QUIC 与 HTTP/3：把可靠传输搬进用户态

QUIC（RFC 9000）：跑在 UDP 之上的通用传输协议，内建：

```
┌──────────────────────────────────────────────┐
│ HTTP/3 (RFC 9114)                            │
├──────────────────────────────────────────────┤
│ QUIC: 流(stream)多路复用 + 可靠传输 + 拥塞控制    │
│       + 连接迁移(connection ID) + 0-RTT 握手   │
├──────────────────────────────────────────────┤
│ TLS 1.3 (内建，不是叠在上面)                     │
├──────────────────────────────────────────────┤
│ UDP (RFC 768)                                │
├──────────────────────────────────────────────┤
│ IP                                           │
└──────────────────────────────────────────────┘
```

维度	TCP + TLS + HTTP/2	QUIC + HTTP/3
建连时延	TCP 1 RTT + TLS 1–2 RTT	1 RTT（首次），**0-RTT**（恢复）

维度	TCP + TLS + HTTP/2	QUIC + HTTP/3
多流 HOL	有（TCP 层）	无（每 stream 独立重传）
加密范围	只加密载荷，TCP 头明文	**头也加密**（除少量必要字段）
中间盒	受 NAT/防火墙“特殊对待”	走 UDP，可能被限速/阻断
连接迁移	四元组变了就断	**Connection ID** 不变，换 Wi-Fi 不断线
拥塞控制	内核实现，升级慢	用户态可插拔（CUBIC/BBR 可换）
CPU 成本	低（有硬件卸载）	高（用户态 + 加密）

QUIC 与 TCP 的本质取舍：QUIC 不是“更好的 TCP”，而是“**把传输层从内核里解放出来**”。代价是失去内核的成熟优化（TSO/GRO、零拷贝）与中间盒友好性；收益是**可演进**（部署新拥塞控制不用等操作系统升级）。

17.5 DNS：Internet 的“电话簿”

项目	内容
端口	53（UDP 为主，响应 >512 B 或区域传送用 TCP）
结构	根 → TLD（`.com` / `.org`）→ 权威服务器，递归解析
记录类型	`A`（IPv4）、`AAAA`（IPv6）、`CNAME`（别名）、`MX`（邮件）、`TXT`、`NS`
缓存	按 TTL 缓存，TTL 短则切换快、查询多
隐私	明文 UDP → **DoT**（853/TCP+TLS）、**DoH**（443/HTTPS）

为什么 DNS 用 UDP：一次查询一次响应，**握手成本占比太高**；丢了自己重传即可（幂等）。

17.6 往下走一层：物理层与弹性缓冲（Elastic Buffers）

Week 8 的《Elastic Buffers》讲的是“**最底层的可靠性抽象**”：物理层把不可靠的信号变成可靠的比特。

问题	机制
收发双方时钟频率不可能完全相同（晶振有 ppm 级误差）	**弹性缓冲（elasticity buffer）** 吸收速率差
无法用反馈（信号级反馈太慢）	用 **MTU**（限制单包长度）+ **包间间隔（inter-packet gap）** 保证缓冲不溢出/不排空
多用户共享介质	TDM（时分）、FDM（频分）、CSMA（载波侦听）

核心不等式（讲义推导的结论形式）：设标称速率 r、时钟容差 ϵ（ppm），最大传输单元 p_{max}，则弹性缓冲大小约需

$$B \approx 2\epsilon r \frac{p_{max}}{r} \cdot \text{（常数因子）} \quad \text{且包间间隔} \geq \frac{B}{2\epsilon r}$$

直觉（讲义原话的意译）：时钟越准，所需的缓冲与包间间隔越小；**理想时钟下两者都不需要**。

> 这一节解释了为什么以太网有“最小帧间隔 96 比特时间”、为什么 MTU 不能无限大、为什么“抖动”在物理层就已经存在。

17.7 常见误解

误解	事实
“HTTP/2 一定比 HTTP/1.1 快”	在**丢包率高**的链路上，TCP 层 HOL 会让 HTTP/2 更慢。
“QUIC 就是 UDP 版 TCP”	QUIC 内建 TLS、多流、连接迁移；它是“传输层 + 安全层”的合并，不是简单复刻。
“DNS 只用 UDP”	大响应、DNSSEC、区域传送都用 TCP；DoH 走 443/TCP。
“0-RTT 没有代价”	有：0-RTT 数据可被**重放**，只能用于幂等请求（如 GET），不能用于支付。
“加密只影响安全，不影响性能”	加密改变中间盒行为（NAT 超时、QoS 识别），QUIC 在部分网络被限速就是例子。

第 18 章 五个 Lab 精讲（check0—check4）

这一章讲“**为什么这样设计**”与“**会在哪里翻车**”；逐行的代码骨架、调试命令与坑清单放在配套文档《CS144_计算机网络_项目上手速查.md》里，避免两份文档重复。

18.1 总览：一条单向依赖链

```
check0  ByteStream ──────────────┐
   │  (push/peek/pop/close/capacity) │
   ▼                               │
check1  Reassembler ── 写 ByteStream ─┘
   │  (insert(first_index, data, is_last))
   ▼
check2  TCPReceiver ── 持有 Reassembler + Wrap32
   │  (receive(TCPSenderMessage) / send() → TCPReceiverMessage)
   ▼
check3  TCPSender ── 读 ByteStream + 定时器 + 重传
   │  (push / receive / tick / make_empty_message)
   ▼
check4  真实网络测量（与代码无关，考的是方法论）
```

18.2 check0：ByteStream 与 webget

两个任务：

1. `webget`：用 `TCPSocket` 发一个 HTTP 请求，读到 EOF 为止（handout 说约 8 行）。
2. `ByteStream`：内存里的**流控字节流**。

ByteStream 的接口契约（handout 原文语义）：

侧	方法	语义
Writer	`push(std::string data)`	只推入**可用容量允许**的部分（可能截断！）
Writer	`close()`	流结束，之后不能再写
Writer	`available_capacity()`	现在还能写多少字节
Writer	`bytes_pushed()`	累计推入字节数（含被截断的？**不，是实际接受的**）
Reader	`peek() → std::string_view`	看接下来的字节（**不消费**）
Reader	`pop(uint64_t len)`	消费 len 字节
Reader	`is_finished()`	已 close **且** 全部 pop 完
Reader	`bytes_buffered()` / `bytes_popped()`	当前缓冲 / 累计消费

三个设计要点：

1. **容量 ≠ 流长度**。handout 特意强调：容量 1 字节的流也能传 TB 级数据，只要写者写一个、读者读一个。
2. **`peek()` 返回 `string_view`**：零拷贝，但**生命周期绑定到内部缓冲**——读者下一次 `pop` 后可能失效。
3. **性能门槛 0.1 Gbit/s**（三种 pop 长度），好的实现能到 10 Gbit/s。这直接决定数据结构选型（见速查文档）。

为什么先做这个：它是后面所有模块的“地基”——Reassembler 往里写，TCPSender 从里读。

18.3 check1：Reassembler

任务：把乱序、可能重叠、可能重复的子串，拼成连续字节流。

```
输入 (first_index, data):
   (4, "efgh")   ← 先到，暂存
   (0, "abcd")   ← 到齐了！0..7 连续 → 全部写入 ByteStream
   (2, "cdef")   ← 与已写入部分重叠 → 只保留 8.. 之后的部分（这里是空）
   (8, "ijkl", is_last=true) ← 标记流结束
```

三类知识（handout 的分类，必须背）：

类别	处理
正好是“下一个字节”	立刻写入 ByteStream

类别	处理
在容量内但前面有洞	暂存在 Reassembler 内部
超出容量	**丢弃**（不存！）

容量是“双重上限”：ByteStream 里缓冲的字节 + Reassembler 里暂存的字节，**合计**不超过 capacity。这是 handout 图上绿+红的总和。

三条硬性要求（评分点）：

1. **不许存重叠数据**（否则“容量=内存上限”被破坏，算 style 违规）；
2. **不许提前写**（前面有洞就不能写）；
3. **超出容量的字节必须丢**（不能“先存着以后用”）。

易错点：`count_bytes_pending()` 只统计**暂存**（未写入 ByteStream）的字节，不含已写入的。

18.4 check2：Wrap32 与 TCPReceiver

两个任务：

1. `Wrap32::wrap/unwrap`（绝对序号 ↔ 32 位序号）；
2. `TCPReceiver`：收段 → 解包 → 推给 Reassembler；回 ackno + window。

TCPReceiver 的状态机（只有 3 个状态）：

```
LISTEN ──收到 SYN──▶ SYN_SENT ──装配到 FIN──▶ FIN_RECV
（无 ISN，ackno 为空）   （有 ackno，流未结束）      （流结束）
```

四个必须处理的细节：

细节	规则
ISN 的确定	**第一个带 SYN 的段**的 seqno 就是 ISN（SYN 可能和数据/FIN 同段）
ackno 何时为空	收到 SYN 之前，`ackno = std::nullopt`（不能编一个 0 出来）
ackno 的值	`wrap(stream_index + 1 + (已收 FIN ? 1 : 0), ISN)`
window 上限	`uint16_t`，最大 65535；`window = capacity - bytes_buffered`（还要扣除暂存）

最易错的一处：`ackno` 指向的是**下一个需要的序号**，而“已经装配到 FIN”时它要**再加 1**（FIN 占一个序号）。

18.5 check3：TCPSender（全 lab 最难）

四个入口：

```
push(transmit)                    ← 尽量填满窗口，发新段
receive(TCPReceiverMessage)       ← 更新窗口、清理已确认段
tick(ms_since_last_tick, transmit) ← 超时判断与重传
make_empty_message()              ← 生成零长度段（用于纯 ACK）
```

必须维护的状态（最小集）：

状态	用途
`next_seqno`（绝对）	下一个要发的序号
`window_size`（默认 1）	接收方通告的窗口
在途队列（按序号有序）	重传时取“序号最小”的那个
定时器（是否运行 + 剩余时间）	只有一个
`RTO`（当前值）与 `initial_RTO`	退避时翻倍，新 ACK 时复位
连续重传计数	供上层判断“连接是否无望”
SYN/FIN 是否已发	避免重复发

三条 handout 明确规定的规则（考试与实现都爱考）：

1. **零窗口时 `push` 假装窗口为 1**（只发 1 字节探测），但**不能真的记住 1**；
2. **窗口满 ≠ 窗口为零**；
3. **每个段必须完整落在窗口内**，且长度 ⩽ `MAX_PAYLOAD_SIZE`。

最容易翻车的三件事：

- **RTO 该不该重置**：只有“推进了 ackno”的新 ACK 才重置；重复 ACK 不重置。
- **定时器该不该重启**：发送新数据时若定时器**未运行**才启动（不是每次发送都重启）。
- **重传哪个段**：序号最小的未完全确认段（不是“最近发的”）。

18.6 check4：测量真实世界

任务：选两条路径（一条 RTT > 100 ms 的远距离路径 + 一条与同学的私网路径），各跑 **⩾1 小时** ping，然后做 13 项分析。

分析项	方法	想说明什么
交付率	收到回复数 / 发出请求数	路径的长期可靠性
最长连续成功 / 最长连续丢失	扫描序号	丢包是独立还是成串
丢包自相关	条件概率 $P(\text{第 } N+k \text{ 个也丢} \mid \text{第 } N \text{ 个丢})$，$k \in [-10, 10]$	丢包是否“突发（bursty）”
min / max RTT	直接取极值	min RTT ≈ 传播时延下限
RTT 时间序列图	横轴真实时刻	看拥塞/路由变化

分析项	方法	想说明什么
RTT 的 CDF	排序后画累积比例	分布形状（长尾？双峰？）
RTT 自相关散点	RTT_N vs RTT_{N+1}	时延的可预测性
吞吐 vs 发送速率	加大包长/频率（<10 s，`-s ≤ 1400`）	找到路径的“膝点”

方法论要点（这才是这个 checkpoint 真正教的东西）：

1. `-D` **加时间戳**：没有时间戳就无法做时间序列与自相关；
2. `-i 0.2` **= 5 包/秒**：handout 明确说不要长期超过这个速率（别把别人网络打爆）；
3. `mtr` **在 VM 里常失效**：handout 建议在宿主机上跑；
4. **丢包要按序号找**：Linux 的 `ping` **不会**为丢失的包打印任何行，必须自己对齐序号；
5. **相关性 ≠ 因果**：RTT 上升可能是排队，也可能是路由变了。

18.7 五个 checkpoint 的共同“元技能”

技能	在哪用到	说明
不变量思维	全部	每个类先写“什么永远为真”（如“暂存区不与已写区重叠”）
边界条件	全部	空流、单字节、窗口为 0、序号回绕、FIN 与数据同段
用测试当规格	全部	`TEST_ONLY` 跑单个用例，失败信息里有测试名与断言位置
抓包验证	check1/3/4	`tcpdump` + Wireshark 看真实字节
写清楚取舍	全部 writeup	评分看“你为什么这样选”，不是“你选了最花的”

第 19 章 与已有笔记的衔接

同目录下已有多份系统类笔记，CS144 与它们的交叉点非常密集。这一章说明“哪些概念可以复用、哪些必须重新建立”。

已有笔记	可直接复用的部分	需要重新建立的部分
CS111 操作系统原理	文件描述符与 socket 的统一抽象、内核/用户态边界、中断与定时器、缓冲区管理	内核网络栈的分层（`sk_buff`、NAPI、软中断）、TCP 的“每连接状态机”规模问题
CSAPP 深入理解计算机系统	字节序、位运算、`struct` 打包与对齐、异常控制流	校验和的反码运算、序列号回绕的模运算、**网络字节序**（大端）的强制约定
CS425 分布式系统	RPC 语义（at-most-once / at-least-once / exactly-once）、超时与重试、幂等键、故障模型	传输层如何**在不可靠信道上**提供“可靠有序字节流”；拥塞控制作为“分布式资源分配”

已有笔记	可直接复用的部分	需要重新建立的部分
CMU 15-418 并行编程	流水线、吞吐 vs 时延、Amdahl 定律、批处理与流水化的取舍	网络里的“流水线”是**窗口**而不是指令流水；瓶颈在**共享链路**而非 ALU
CS106L Modern C++	RAII、`std::string_view`、`std::optional`、移动语义、`const` 正确性	零拷贝的**生命周期陷阱**（`string_view` 指向内部缓冲）

三个特别值得点出的“跨课程共鸣”：

1. **可靠传输 ≈ 分布式系统的“恰好一次”**。CS425 讲 RPC 重试要幂等键；CS144 讲 `push()` 要变成幂等操作靠“序号 + 累计确认”。**同一枚硬币的两面**：一个在应用层做，一个在传输层做。
2. **拥塞控制 ≈ 分布式资源分配**。Week 5 讲义原话：“在网络里，几乎所有涉及**去中心化资源分配**的问题都叫拥塞控制。”这和阿罗–德布鲁均衡、α–fairness 是同一个数学对象。
3. **缓冲区 ≈ 操作系统的页缓存**。CS111 的“缓冲区大小决定吞吐与延迟”，在网络上就是“窗口大小 + 队列长度”：**缓冲越大吞吐越高、时延越大**（bufferbloat），取舍逻辑完全一致。

第 20 章 复习清单与自测题

20.1 必须能“不看书画出来”的 10 张图

#	图	检验标准
1	协议栈封装图（HTTP/TCP/IP/Ethernet）	能标出每层头部长度与 MSS 的来源
2	TCP 头（20 字节，逐字段）	能说出每个字段的位宽与作用
3	三种序号对照表（seqno/absolute/stream index）	能用 `cat` 例子手算
4	滑动窗口（左边界/右边界/已发未确认）	能画出“收到 ACK 后窗口如何滑”
5	三次握手 + 四次挥手时序	能标出每个段的标志位与序号消耗
6	TCP 状态机（11 状态）	能说出 CLOSE_WAIT 与 TIME_WAIT 的成因
7	AIMD 锯齿图 + 慢启动	能标出 ssthresh 与两个阶段
8	CUBIC 的三次曲线（凹/平台/凸）	能解释为什么“平台区”要慢
9	最长前缀匹配的例子	给定 4 条表项与 1 个地址，能选对出端口
10	令牌桶 + WFQ 的时延上界	能写出 $\sigma + \rho t$ 与 $\sum(p/r + l/c) + 3\sigma/\rho$

20.2 自测题（12 题，答案在后）

1. 一条 1 Gb/s、RTT 80 ms 的路径，BDP 是多少字节？要跑满需要多大的窗口（以 1460 B 的 MSS 计，约多少个段）？
2. 若 ISN = $2^{32}-3$，字节流是 `"hi"`，请写出 SYN、`h`、`i`、FIN 的 seqno、absolute seqno 与 stream index。
3. 接收方收到 `seqno=500, len=100`，但它期待的是 400。它会回什么 ackno？这会触发发送方的什么行为？
4. 为什么 TCP 的 RTO 要“均值 + 4 倍偏差”，而不是“均值的 2 倍”？
5. 一个段被重传过，随后收到它的 ACK。按 Karn 算法，能否用这次测量更新 SRTT？为什么？
6. `cwnd=16 MSS`，一个 RTT 内收到全部 16 个 ACK（拥塞避免阶段）。下一个 RTT 开始时 `cwnd` 是多少？
7. 发生超时后，`ssthresh` 与 `cwnd` 分别变成什么？和“3 个 dupACK”有何不同？
8. 为什么 TIME_WAIT 要等 2×MSL 而不是 1×MSL？
9. HTTP/2 已经多路复用了，为什么还会被 TCP 的队头阻塞影响？
10. 给定转发表 `0/0→A`、`10/8→B`、`10.1/16→C`、`10.1.2/24→D`，目的地址 `10.1.2.3` 与 `10.2.3.4` 分别走哪个口？
11. NAT 后面的主机为什么不能直接被外网连接？两种解决思路分别是什么？
12. 为什么 BBR 在“有随机丢包的无线链路”上通常优于 CUBIC？

20.3 答案

1. $BDP = 10^9 \times 0.08/8 = 10$ MB。段数 $= 10 \times 10^6/1460 \approx 6849$ 个 MSS。（这说明为什么需要窗口缩放：16 位窗口最大仅 65535 B。）
2. ISN $= 2^{32}-3$：SYN 的 seqno $= 2^{32}-3$，absolute $= 0$；`h` 的 seqno $= 2^{32}-2$，absolute $= 1$，index $= 0$；`i` 的 seqno $= 2^{32}-1$，absolute $= 2$，index $= 1$；FIN 的 seqno $= 0$（回绕），absolute $= 3$。
3. 回 `ackno=400`（累计确认只报到第一个缺口）。发送方会把它当作**重复 ACK**；若累计收到 3 个，触发**快速重传** 400 段。
4. RTT 分布是**长尾**的：均值 + 常数倍会低估尾部，导致伪重传。用“均值 + 4×平均绝对偏差”近似“均值 + 4σ”，能覆盖绝大多数样本，同时不至于过大。
5. 不能。重传造成“ACK 对应原始还是重传”的歧义（retransmission ambiguity），Karn 算法规定**跳过该样本**，并且每次重传让 RTO 翻倍。
6. $16+1=17$ MSS（拥塞避免每 RTT +1）。注意每个 ACK 只加 $MSS^2/cwnd$，16 个 ACK 合计 +1 MSS。
7. 超时：`ssthresh = max(FlightSize/2, 2MSS)`，`cwnd = 1 MSS`，进入**慢启动**。3 dupACK：同样的 `ssthresh`，但 `cwnd = ssthresh + 3MSS` 并进入**快速恢复**（随后回落到 `ssthresh`）。
8. 因为要覆盖“**我方最后一个 ACK 丢失 → 对方重传 FIN → 我方重发 ACK**”这一整个往返，以及“对方重传的 FIN 在网络中消亡”的时间；1×MSL 只够单向。

9. 因为多路复用跑在**一条 TCP 连接**上，而 TCP 保证**按序交付**；任一 TCP 段丢失，后续所有段（无论属于哪个 stream）都要等它重传，形成 TCP 层 HOL。
10. `10.1.2.3` → **D**（/24 最长）；`10.2.3.4` → **B**（匹配 /8，但不匹配 /16 与 /24）。
11. 因为内网地址在公网**不可路由**，且 NAT 只在“内网先发包”时建立映射。两种思路：① **端口映射/UPnP**（在 NAT 上静态开洞）；② **打洞**（STUN 发现公网地址 + 会合服务器交换 + 同时发包），失败则回退 **TURN 中继**。
12. 因为 BBR 用“交付速率 + min RTT”建模，**不把丢包当拥塞信号**；无线随机丢包不会让它误降窗口，而 CUBIC 会把每次丢包当成拥塞而减窗。

20.4 考前 3 小时清单

- [ ] 手推一遍“三种序号”的换算（含回绕）
- [ ] 手画 TCP 状态机，标出 11 个状态与 4 条关键边（SYN、FIN、ACK、RST）
- [ ] 背 RTO 公式（SRTT/RTTVAR/α/β）与 Karn 算法
- [ ] 背 AIMD 的逐 ACK 更新式与 ssthresh 规则
- [ ] 背 CUBIC 的 $C = 0.4$、$\beta = 0.7$；知道 BBR 不是 RFC
- [ ] 背 WFQ 时延上界与令牌桶 (σ, ρ)
- [ ] 过一遍 4 套历年题（`exams/*.pdf`），只看错题
- [ ] 检查 cheat sheet：公式 + 状态机 + 一张 TCP 头

第 21 章 参考来源

21.1 课程官方材料（均已实际抓取，访问日期 2026-11）

材料	URL	状态
课程主页（Fall 2026）	https://cs144.github.io/	✅ 抓取成功
课程大纲/评分/迟交/诚信	https://cs144.github.io/logistics.pdf	✅ 文本已提取
Lab FAQ	https://cs144.github.io/lab_faq.html	✅（部分条目仍用旧版 API 名，见 21.4）

材料	URL	状态
check0 handout	https://cs144.github.io/assignments/check0.pdf	✅
check1 handout	https://cs144.github.io/assignments/check1.pdf	✅
check2 handout	https://cs144.github.io/assignments/check2.pdf	✅
check3 handout	https://cs144.github.io/assignments/check3.pdf	✅
check4 handout	https://cs144.github.io/assignments/check4.pdf	✅
check5–check7 handout	`assignments/check5.pdf` 等	❌ HTTP 404（本学期未公开）
讲义：课程介绍	https://cs144.github.io/handouts/week-1-logistics.pdf	✅
讲义：可靠性	https://cs144.github.io/handouts/week-1-reliability.pdf	✅
讲义：TCP	https://cs144.github.io/handouts/week-2-tcp.pdf	✅
讲义：分组交换 I	https://cs144.github.io/handouts/week-3-packet-switching-a.pdf	✅
讲义：为什么拥塞控制	https://cs144.github.io/handouts/week-4-why_congestion_control.pdf	✅
讲义：什么是拥塞控制	https://cs144.github.io/handouts/week-4-what_congestion_control.pdf	✅
讲义：怎么做拥塞控制	https://cs144.github.io/handouts/week-4-how_congestion_control.pdf	✅
讲义：路由导论	https://cs144.github.io/handouts/routing-intro.pdf	✅
练习题与答案（4 套 8 个 PDF）	统一路径前缀 `https://cs144.github.io/exams/`（无目录索引，需直接访问文件名，例：https://cs144.github.io/exams/21fa-midterm.pdf ）。文件名：`21fa-midterm.pdf`、`21fa-midterm-ans.pdf`、`21fa-final.pdf`、`21fa-final-ans.pdf`、`sp23_midterm.pdf`、`sp23_midterm-ans.pdf`、`sp23_final.pdf`、`sp23_final-ans.pdf`	✅ 均已验证可下载（本文未逐题引用）

21.2 Google Docs 形式的 lecture notes（经 `export?format=txt` 取得正文）

课程主页把 11 篇 lecture notes 放在 Google Docs 上。直接 `curl` 页面 HTML 拿不到正文，但改用导出接口可获得纯文本：

```
https://docs.google.com/document/d/<DOC_ID>/export?format=txt
```

#	主题（据正文首行判断）	URL
1	Honor Code 边界案例（答疑汇编）	https://docs.google.com/document/d/18hywKDuHFYscoMjId58KZqiFdvroiyz_zQjkGpvCEQU
2	Datagrams, encapsulation, and multiplexing	https://docs.google.com/document/d/1uODk7ESsFE9Fz_aUaovu0gGX4rsb4WM-UXgoe5RoYfo
3	UDP / 端口 / 协议栈（UDP 之上能建什么）	https://docs.google.com/document/d/1s6_yZ6zCZqw18q4mZbQXgXmTkufTDDgNYUhIgY5Kwu8
4	TCP I / Reliability from unreliability（含三次握手推导）	https://docs.google.com/document/d/1VlnHwTMV_sY9u3EoKrC_JG4saDE5YM9n-RGETN4e9UE
5	Packet Switching（时延公式、排队）	https://docs.google.com/document/d/1bgj02U2J6MGObNMg0z7ajPSK2JlmRgSJO5OFwY-b_cc
6	拥塞控制：模型与崩溃（collapse）	https://docs.google.com/document/d/1e7A2xa_gAaFuZqJtQk6hE0gfMXYV_gO8l_AcXycMI1g
7	拥塞控制：公平性与效用函数	https://docs.google.com/document/d/1uCFpj9ttiqNYESQ1cNSWwcAdn_JiRj3C8-pmkrxoN6I
8	The eras tour of home networking（Level 0–8）	https://docs.google.com/document/d/1B31j3miOBaLSSLU-uvD5Z2wFjvHiCrScI5OI7HhWIv0

#	主题（据正文首行判断）	URL
9	续：NAPT、P2P、STUN/TURN（Level 8–9d）	https://docs.google.com/document/d/1Yx1ZIVUGx2uRy_Vi6WICisM4nKurRNlk6Kn9bdMIG_8
10	Physical Layer / Elastic Buffers	https://docs.google.com/document/d/1wl5IYYobn9quPmr_ViPF2D94OWtPA9eWO-TPmFZXxXM
11	Internet Security	https://docs.google.com/document/d/1R6aL2_ZNAR-oxFNpL6mtOlmfiMgtRDMqkXNJYeYEVAg

> 说明：#3 与 #4 在主页上对应不同的讲次链接，但导出内容的开头高度重叠（都以“best effort 的后果”起头），笔者未能完全区分二者边界，**按主题归并**如上。这是 21.4 节“最不确定”的第一条。

21.3 教材与标准

来源	URL	用途
Kurose & Ross, *Computer Networking: A Top-Down Approach* (8th ed.)	出版信息见课程主页链接	传输层/网络层/链路层体系（第 3–6 章）
Peterson & Davie, *Computer Networks: A Systems Approach*	https://book.systemsapproach.org/	全文免费；拥塞控制与路由章节
Kaashoek & Saltzer, *Principles of Computer System Design* 第 7–8 章	课程主页列为可选教材（MIT OCW 有全文 PDF）	分层、端到端原则、命名
RFC 9293（TCP，2022，取代 RFC 793）	https://www.rfc-editor.org/rfc/rfc9293.txt	TCP 头、状态机、RTO 引用
RFC 5681（TCP Congestion Control）	https://www.rfc-editor.org/rfc/rfc5681.txt	慢启动/拥塞避免/快速重传的规范表述
RFC 6298（Computing TCP's RTO）	https://www.rfc-editor.org/rfc/rfc6298.txt	SRTT/RTTVAR/Karn
RFC 6675（SACK loss recovery）	https://www.rfc-editor.org/rfc/rfc6675.txt	SACK 恢复算法
RFC 7323（Window Scale / Timestamps）	https://www.rfc-editor.org/rfc/rfc7323.txt	窗口缩放、PAWS
RFC 9438（CUBIC）	https://www.rfc-editor.org/rfc/rfc9438.txt	CUBIC 的 C、β 与曲线

来源	URL	用途
RFC 3168（ECN）	https://www.rfc-editor.org/rfc/rfc3168.txt	ECN/ECE/CWR
RFC 9000（QUIC）	https://www.rfc-editor.org/rfc/rfc9000.txt	QUIC 传输
RFC 9114（HTTP/3）	https://www.rfc-editor.org/rfc/rfc9114.txt	HTTP/3 映射
RFC 1035（DNS）	https://www.rfc-editor.org/rfc/rfc1035.txt	DNS 报文格式
RFC 8484（DoH）	https://www.rfc-editor.org/rfc/rfc8484.txt	加密 DNS
RFC 3489（经典 STUN/NAT 分类）	https://www.rfc-editor.org/rfc/rfc3489.txt	四种 NAT 类型
RFC 8489（STUN）/ 8656（TURN）	https://www.rfc-editor.org/rfc/rfc8489.txt；https://www.rfc-editor.org/rfc/rfc8656.txt	打洞与中继

21.4 未能获取 / 存在不确定性的材料

1. **check5 及之后的 handout**：`assignments/check5.pdf`、`check6.pdf`、`check7.pdf` 全部 **404**；主页 HTML 中相关条目被注释掉。本文档因此只覆盖 check0–check4。
2. **Google Docs 的讲次归属**：11 篇文档中，#3 与 #4 的边界、以及主页“Week 4/5”表格里若干条目的对应关系存在重叠，笔者按正文内容归并，可能与课程原始排布略有出入。
3. **lab FAQ 的时效性**：FAQ 中部分问答仍使用旧版 starter code 的 API 名（如 `TCPReceiver::segment_received()`、`TCPSender::ack_received()`、`WrappingInt32`、`tcp_state.hh`、`minnow/tests`），而 Fall 2026 的 starter code 名为 **lmp**（`git clone https://github.com/cs144/imp`），API 已改为 `receive()/send()/push()/tick()`、`Wrap32`。本文档以 **check0–check4 handout 的 API** 为准，FAQ 仅用于理解语义。
4. **验收日期与比例**：本文引用的日期（10/28 期中、12/9 期末、各 checkpoint due date）与评分比例均来自 2026-11-01 抓取的页面快照，**以课程当学期主页为准**。
5. **硬件/内核具体数值**：文中的 Linux 默认值（如 IW=10、`tcp_fin_timeout=60s`、延迟 ACK 40 ms）来自通用工程经验，**非 CS144 材料原文**，属“背景补充”。

> **备注**：本文中的 ASCII 图均按课程讲义幻灯片与 RFC 9293 的图重绘；代码片段为教学用简化伪代码，**不是**可直接提交的 lab 解答。建议配合原始材料学习：课程主页 https://cs144.github.io/ 、Peterson & Davie 免费全文 https://book.systemsapproach.org/ 、RFC 9293 https://www.rfc-editor.org/rfc/rfc9293.html 。