Generated on 2025-11-13

ParaN3xus's Blog 2025

Archive of Blog posts in 2025

目录

2025-11-08

DIY Smol Slime 追踪器

我 DIY Smol Slime 经历的记录.

一直很羡慕能在游戏里随便摆 Pose 的好友们, 但是我没钱, 买不起成品 tracker, 又不会焊接, 所以没法 DIY 史莱姆.

今年早些看了一个 Bilibili 视频介绍的小史莱姆(Smol Slime), 狠狠心动了. 正好这学期学校有一整周的实训课程教我们怎么焊接电路板(一周前我还对焊接一无所知, 现在也只是焊接新手), 那就趁这个机会试试吧.

1.1 成果

- 数量: 十点
- 电池: 120 mAh, 提供约 24h 续航
- 尺寸: tbd, 但是十分小巧轻薄
- 追踪效果: tbd

1.2 成本

类别	物品	数量	价格	备注
工具	黄花电烙铁 EP-916L	1	115	趁手即可
	卡诺宇恒温加热台 1010	1	73.01	趁手即可
	得力水口钳	1	13.5	剪去多余的排针用, 趁手即可
	镊子套装	1	18	主要是 SMT 贴片时夹取和放置元件用, 趁手即可
	定制 SMT 激光钢网 10cm*10cm	1	15	
	优利德万用表 UT33A+	1	82.78	最低要求是能测电阻
	迷你锡膏印刷台	1	25.8	闲鱼
	合计		**343.09**	
材料	凯利顺锡浆 50 克	1	25.5	
	焊锡丝	0	0	买电烙铁送的
	两米长三厘米宽松紧带	3	13.12	
	1007 电子线 5 卷共 100 米	1	36.45	明显买多了
	子母扣	9	2.97	
	合计		**128.04**	
电子元件	KEY SMD 小乌龟轻触开关	10	0.48	
	401230 锂电池 3.7V 120mAh	10	7.23	
	贴片电容 CL10B104KC8NNNC 100nF	50	0.0372	

	贴片电容 CL10A225KO8NNNC 2.2uF	50	0.0381	
	贴片电阻 0603WAF1002T5E 2.2uF	100	0.0059	
	有源晶振 YF4032K76833T8188081	10	2.37	
	姿态传感器 ICM-45686	10	19.83	
	3D 磁传感器 QMC6309	10	2.2065	可选, 现阶段小史莱姆固件还不支持磁传感器
	合计		**325.52**	
模块	Holyiot-21017-nRF52840 接收器	1	99	
	Promicro NRF52840 开发板	10	11.89	
	ICM-45686 模块 PCB	10	0	嘉立创免费打样
	十口 USB 分线器	1	26	
	USB-C 公口对 USB-A 公口转换器	10	5.95	
	GoPro 用双肩胸带	1	15.3	挂接胸部 tracker 用
	3D 打印外壳	10	3.54	
	3D 打印挂载托盘	9	1.91	
	3D 打印 GoPro 胸带挂载托盘	1	2.11	
	合计		**373.4**	
合计			**1170.05**	

前面提到的视频中的商品 Styria Mini tracker 十点的价格是 1888. 虽然没有触点充电, 一键唤醒等功能, 但价格便宜 700 多元(甚至还留下了价值 300 多元的不动产)让我感到满意.

如果你决定要跟随本教程制作 tracker, 建议先通读一遍本教程, 明确自己需要准备什么工具或材料, 然后再进行准备和实操.

1.3 过程

我的 Smol Slime tracker 整体上参考官方文档的 DIY 方案, 但是替换了官方方案中使用的成品 ICM-45686 模块, 因此步骤上要再多一条制作该模块.

整体过程如下
1. 焊接 ICM-45686 模块
2. 焊接 tracker
3. 刷写固件, 配对 tracker 和接收器
4. 组装 tracker 与外壳, 绑带
5. 调试, 测试

1.3.1 焊接 ICM-45686 模块

这个步骤是精细操作要求最高的部分. 不使用成品模块能省下约 100 人民币, 但是会额外消耗你约 5 小时时间(我在熟练后需要约 30 分钟来焊接一个该模块). 所以如果你不缺这 100 人民币, 你可以跳过, 并直接购买成品模块.

操作步骤如下

1. 将钢网和 PCB 板安装在夹具上

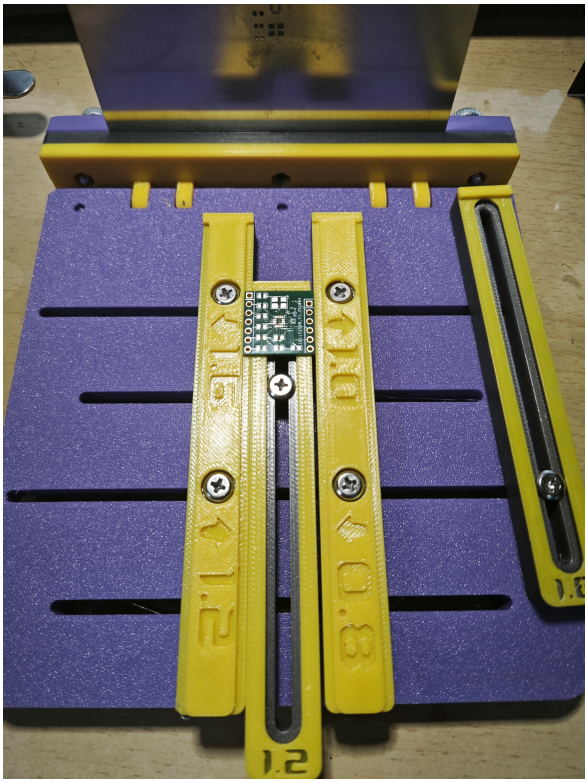

图 1　钢网和 PCB 板安装在夹具上

2. 盖上钢网, 对齐钢网和 PCB

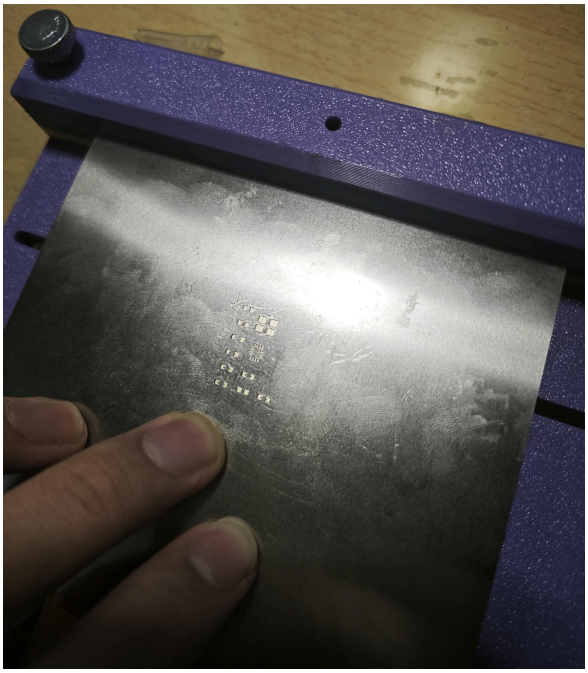

图 2　对齐钢网和 PCB

3. 刷上锡浆. 在覆盖焊点的基础上, 用最少的次数刷上恰好足够的锡浆. 尤其是对于 ICM-45686 和 QMC6309 两个芯片的焊盘, 能刷一次绝不刷两次, 以防止锡浆过多导致焊接后连锡. 刷的时候要用力按平钢网, 防止锡浆渗出到钢网另一侧与 PCB 之间的空隙, 导致连锡.

图 3　刷上锡浆

4. 揭开钢板. 揭开前保证没有大片锡膏在焊点上, 否则揭开后会出现拉尖. 揭开时, 注意从下往上慢慢揭开, 防止锡浆偏移.

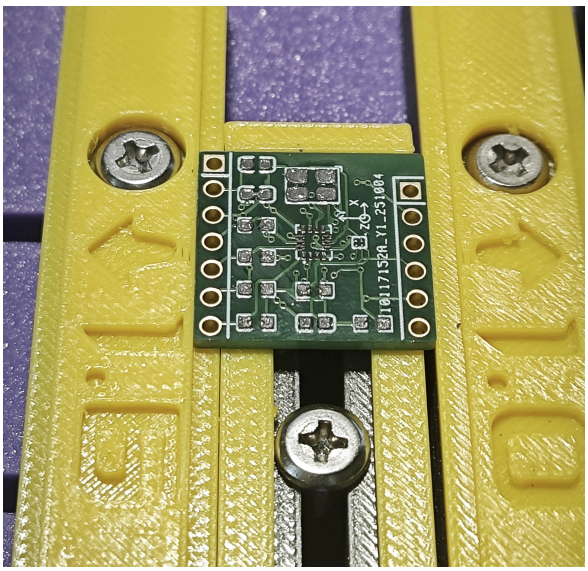

图 4　刷好锡浆的 PCB

注意检查各个焊盘上是否有锡浆, 是否有连锡. 电阻电容和晶振焊盘较大, 轻微连锡无关紧要, 但是 ICM-45686 焊点比较密集, 需要重点关注. QMC6309 焊点较小, 经常出现钢网被堵塞, 锡浆刷不上的情况, 不过因为 QMC6309 是可选安装的芯片, 所以如果你不想在这上面花时间, 可以直接不予理睬.

5. 摆放元器件. 用镊子把元器件摆到 PCB 上. 其中
   - 左上方两个是 100nF 电容的焊盘, 没有正反
   - 下方中间一个是 2.2uF 电容的焊盘, 没有正反
   - 其余小长方形是 10k 电阻的焊盘, 字面朝上即可
   - 中间上方面积最大的是晶振的焊盘, 反光下可见芯片上有一个 “·”, 注意这个 “·” 和焊盘右上角处的 “¬” 在同一方向

- 中间焊点最多的是 ICM-45686 芯片的焊盘, 反光下可见芯片上有一个 “·”, 注意这个 “·” 和焊盘左下角处的 “¬” 在同一方向
- (可选安装, 不影响功能)右侧最小的正方形是 QMC6309 芯片的焊盘, 反光下可见芯片上有一个 “·”, 注意这个 “·” 和焊盘右上角处的 “¬” 在同一方向.

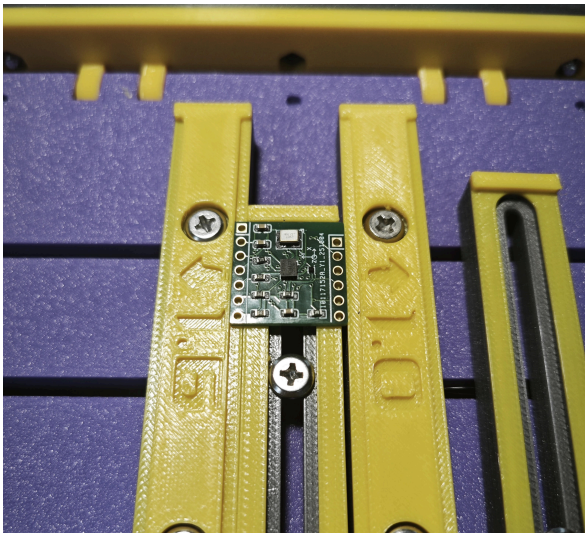

图 5　放好元件的 PCB

6. 加热台打开, 温度设置为 160, 到达后将 PCB 板放上, 预热约 30 秒.

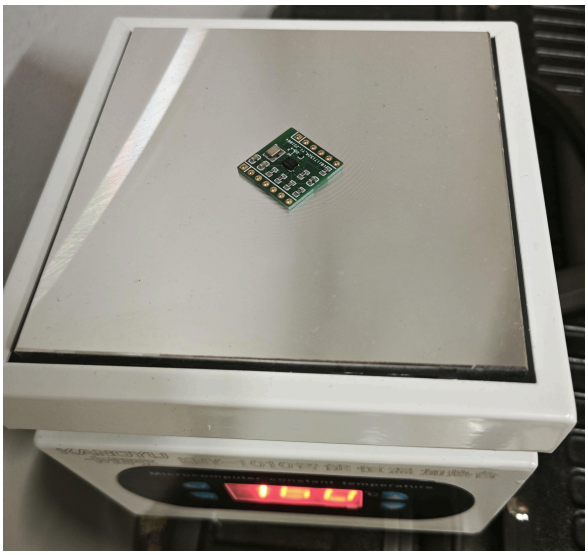

图 6　预热 PCB

背景图是我室友的行李箱. 请不要担心, 因为加热台只有上面的表面是热的. 本教程中没有任何室友或室友的行李箱受到损害.

7. 预热完成后, 调节温度到比锡浆熔点略高. 对于我来说, 这个温度是 210. 然后等待升温后锡浆融化.

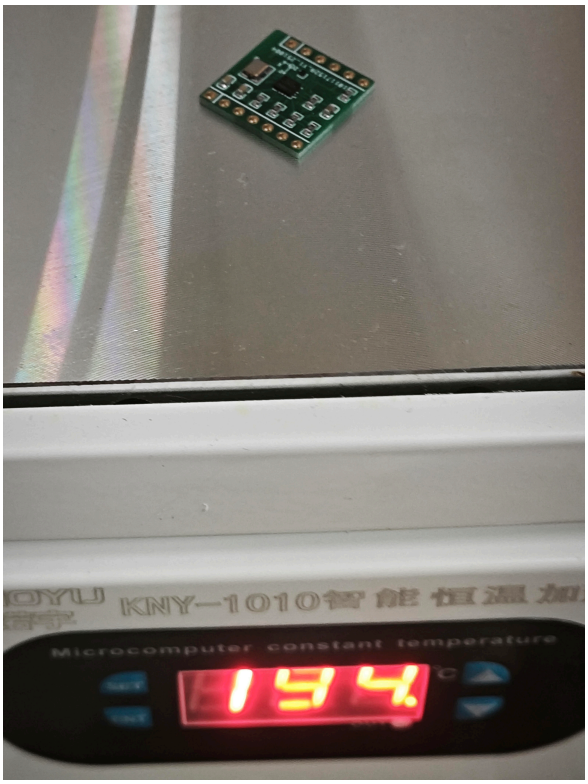

图 7　锡浆融化了

8. 当所有焊点的锡浆已经充分融化(往往是你设置的温度已经到达时), 用镊子直接把 PCB 从加热台上移除, 然后放到一个合适的地方散热(比如钢网上). 关闭加热台(这个加热台没有主动制冷功能). 注意此后的相当长时间内, 加热台仍然是烫的, 请注意安全.
9. 使用万用表检测各焊点是否有连锡或虚焊. 由于写教程时, 我的每个模块都已经被做成了 tracker, 所以无法提供各个测量点的具体参考值, 只能给出检测步骤和大致结果.
   - ICM-45686 连锡检测
     - OSDO 和 SC0: 应为开路
     - CLK 和 SC0: 不应短路
     - CLK 和 +3V3: 不应短路
     - GND 和 +3V3: 不应短路
     - INT1 和 SCX: 不应短路
     - SDX 和 SCX: 不应短路
     - SDX 和 SD0: 不应短路
     - SDA 和 SCL: 不应短路
     - CS 和 SCL: 不应短路
   - ICM-45686 虚焊检测
     - 测不了, 只能祈祷正确
   - 其它
     - 如果不放心, 可自行参考 PCB 设计图进行检测
10. 大功告成

1.3.2 焊接 tracker

这一部分和官方教程基本一致, 但是如果你选择自制 ICM-45686 模块, 那么得益于该模块的尺寸设计, 这一步会相对简单.

1. 给模块背面贴上胶带, 防止其与开发板密切接触导致的短路

图 8　贴好胶带的模块

2. 掰一段六个的排针, 用钳子处理一下, 把外壳移动到一端

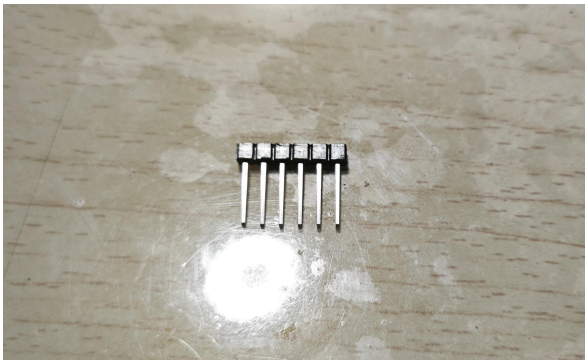

图 9　处理好的排针

再取两个排针, 直接移除外壳, 取出金属排针部分

3. 把开发板, 模块, 排针摆放到图示位置(开发板四角的四组三个排针是起支撑固定作用的, 并不需要焊接)

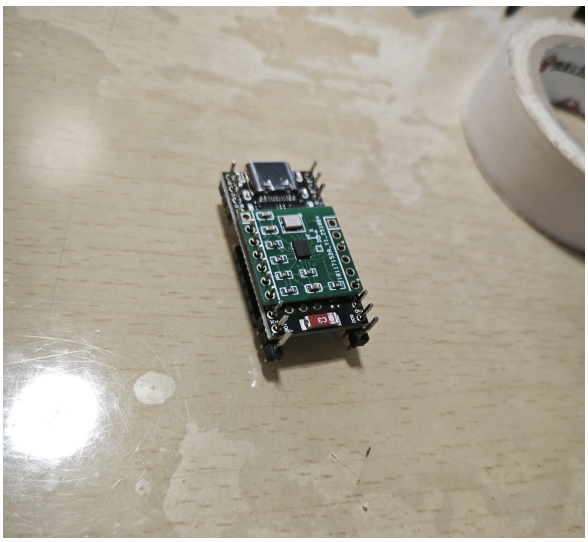

图 10　等待焊接正面的 tracker

4. 焊接正面

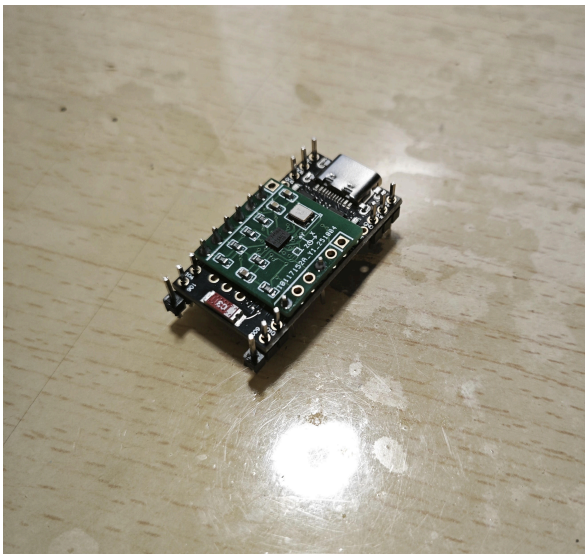

图 11　正面焊接后的 tracker

5. 移除六个排针的外壳, 准备焊接反面

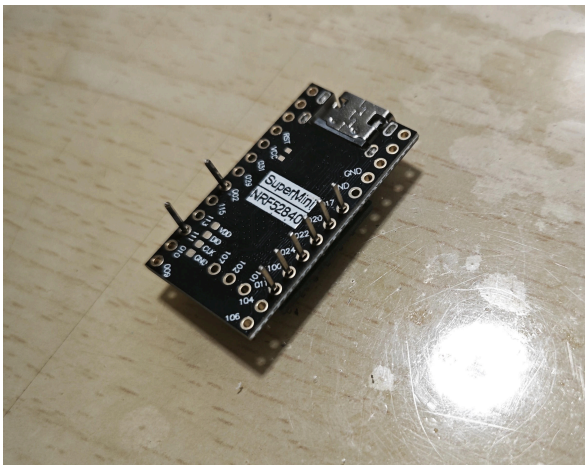

图 12　等待焊接反面的 tracker

6. 焊接反面

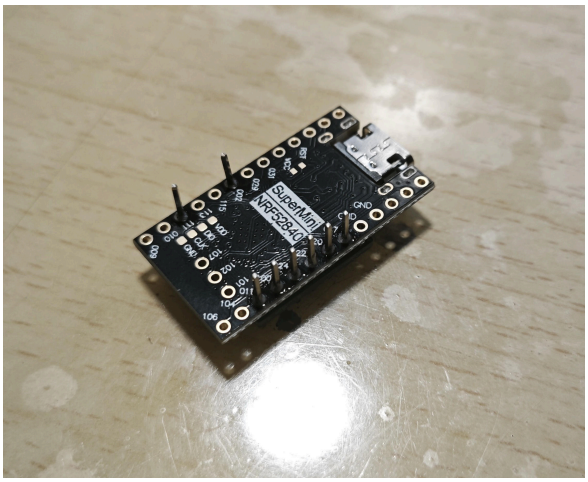

图 13　反面焊接后的 tracker

7. 用钳子剪去多余的排针
8. 焊接电池. 在进行这一步时, 你的 tracker 上就会有指示灯亮起了, 这是正常的, 请不要担心

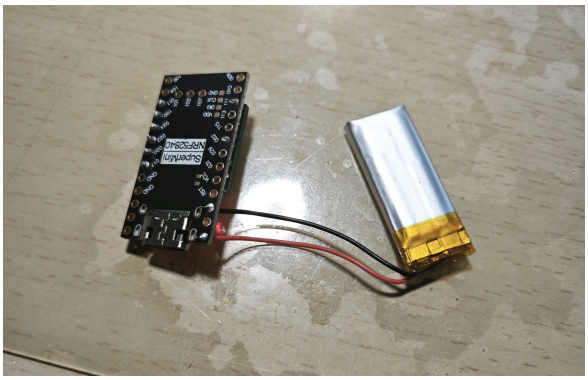

图 14　焊接电池后的 tracker

9. 焊接天线.
   1. 剪取一段和 tracker 长度相当的导线, 剥去一端的一小段线皮(5 毫米已经足够), 把铜线捻在一起(如果你的导线是多股的), 并使用烙铁镀锡.
   2. 在 tracker 的图示位置预先上锡

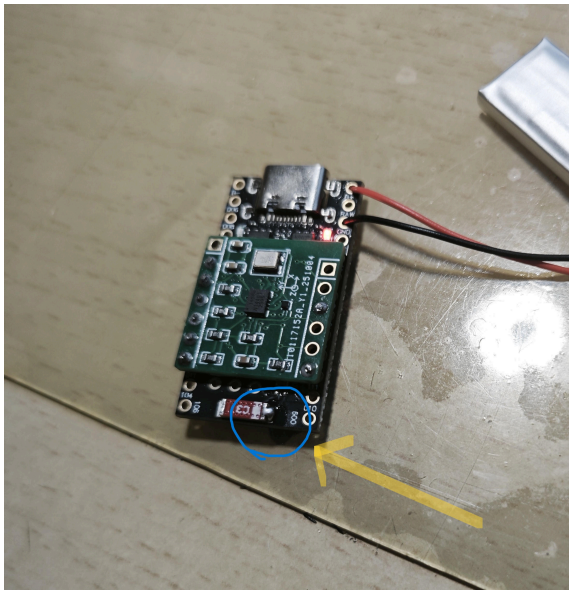

图 15　等待焊接天线的 tracker

   3. 在图示位置焊接天线

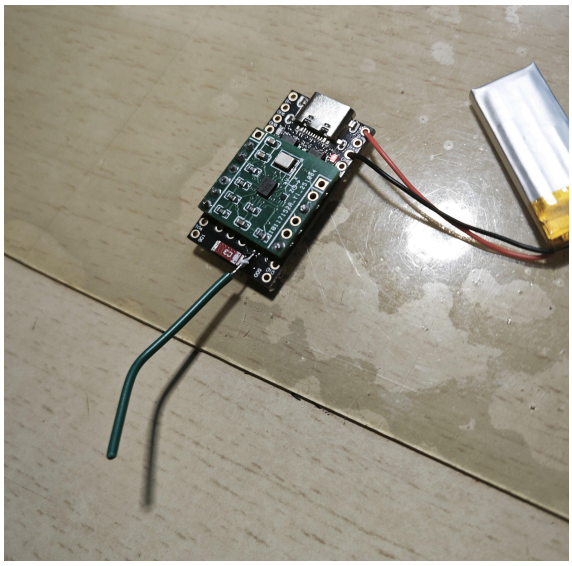

图 16　天线焊接后的 tracker

10. 焊接按钮
    1. 在 tracker 的图示位置预先上锡

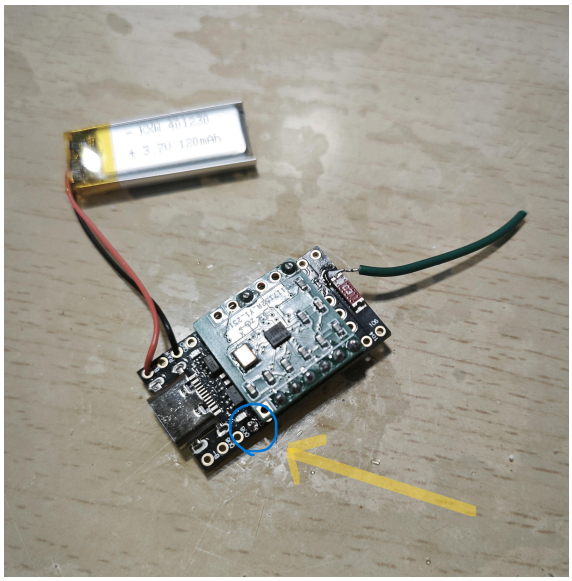

图 17　等待焊接按钮的 tracker

    2. 在图示位置焊接按钮

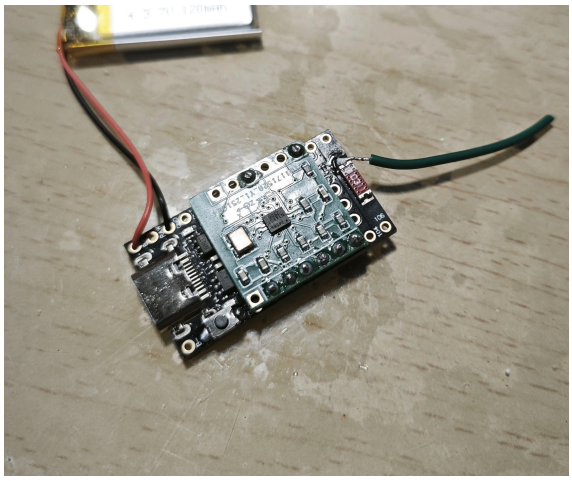

图 18　按钮焊接后的 tracker

11. 大功告成

1.3.3 刷写固件, 配对 tracker 和接收器

在这一步, 你通常就能知道你的 tracker 是否正常工作了.

1. 刷入接收器固件
   1. 下载接收器固件
   2. 安装 nRF Connect for Desktop, 在里面安装 “Programmer” 并打开
   3. 将 Holyiot-21017-nRF52840 接收器连接到电脑. 该接收器还附赠一块磁铁, 把这块此贴在指示灯附近以进入 DFU 模式
   4. 在 Programmer 中选择接收器设备, 刷入接收器固件
2. 刷入 tracker 固件
   1. 下载 tracker 固件. 这里的链接是官方推荐的 ProMicro 设备 SPI 协议有时钟无 WOM 有按钮版本, 如果需要其他版本, 请参阅官方文档
   2. 将 tracker (通过 USB 线)连接到电脑, 应当看到一个名称为 “NICENANO” 的存储设备.
   3. 直接将固件复制到该设备, 稍等片刻后该设备消失, 刷写完成

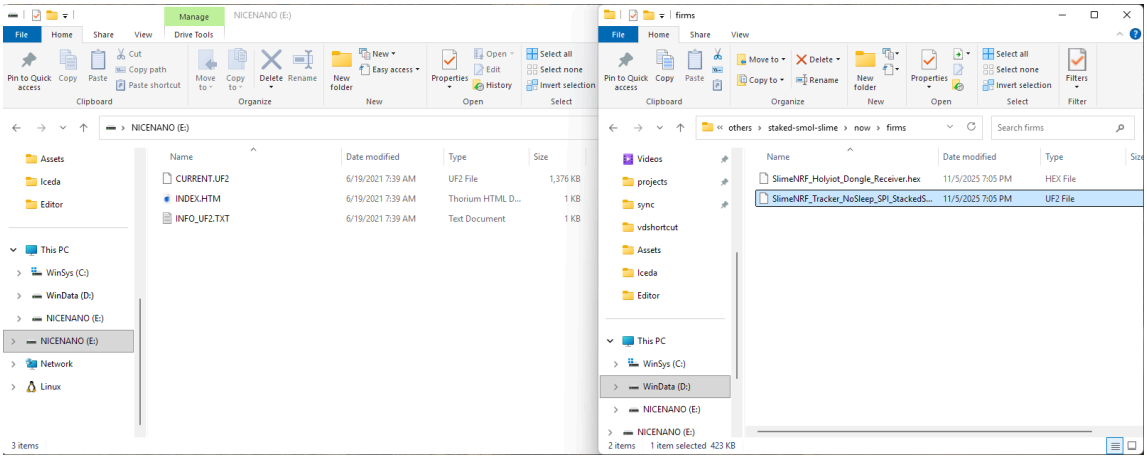

图 19　刷入 tracker 固件

3. 配对 tracker 和接收器
   1. 下载 SmolSlimeConfigurator, 并打开两个实例
   2. 将接收器连接到电脑, 刷新一个 SmolSlimeConfigurator 实例的 COM 接口列表, 应该能看到接收器对应的 COM 接口. 选中该接口, 点击连接, 能看到类似如下输出

```
Connected to COM5
*** Booting SlimeVR-Tracker-nRF-Receiver v0.6.9-c0c2784b78b0 ***
*** Using nRF Connect SDK v3.1.1-cb47ad580360 ***
*** Using Zephyr OS v4.1.99-ff8f0c579eeb ***
[00:00:00.000,305] <inf> hid_event: HID Device: dev 0x1854c
[00:00:00.006,988] <inf> esb_event: 10/256 devices stored
[00:00:00.007,019] <inf> esb_event: Initializing ESB, RX mode
[00:00:00.109,283] <inf> hid_event: New protocol: report
*** Holyiot SlimeNRF Receiver Holyiot-21017 ***
SlimeVR-Tracker-nRF-Receiver 0.6.9+0 (Commit c0c2784b78b0, Build 2025-11-05
02:23:01)
info                        Get device information
uptime                      Get device uptime
list                        Get paired devices
reboot                      Soft reset the device
add <address>               Manually add a device
remove                      Remove last device
pair                        Enter pairing mode
exit                        Exit pairing mode
clear                       Clear stored devices
d
```

由于我已经配对过, 所以这里会显示 `esb_event: 10/256 devices stored`.

3. 这个 SmolSlimeConfigurator 实例切换到 Receiver 选项卡. 在此选项卡下, 点击 “Pairing Mode” 即可进入配对模式, 点击 “⊠ Pairing Mode” 即可退出配对模式.
4. 配对一个 tracker
   1. 连接一个 tracker 到电脑, 刷新另一个 SmolSlimeConfigurator 实例的 COM 接口列表, 应该能看到 tracker 对应的 COM 接口. 选中该接口, 点击连接, 能看到类似如下输出

      ```
      Connected to COM23
      [19:58:33.350,463] <inf> status: USB connected
      [19:58:33.350,494] <inf> status: Status: 8
      [19:58:33.406,372] <inf> status: Charger plugged
      [19:58:33.406,372] <inf> status: Status: 24
      [19:58:33.406,402] <inf> power: Change to battery SOC: 26.55% -> 100.00%
      [19:58:33.406,433] <inf> battery_tracker: Tracker reset
      *** SlimeVR SlimeNRF Tracker ProMicro ***
      SlimeVR-Tracker-nRF 0.6.9+0 (Commit 5d572a0056f3, Build 2025-11-05 02:26:37)
      info                        Get device information
      uptime                      Get device uptime
      reboot                      Soft reset the device
      battery                     Get battery information
      scan                        Restart sensor scan
      calibrate                   Calibrate sensor ZRO
      6-side                      Calibrate 6-side accelerometer
      set <address>               Manually set receiver
      pair                        Enter pairing mode
      clear                       Clear pairing data
      dfu
      ```

   2. 点击 Info 按钮, 查看该 tracker 的地址

      ```
      >>> info
      info
      SlimeVR SlimeNRF Tracker ProMicro
      ```

```
SlimeVR-Tracker-nRF 0.6.9+0 (Commit 5d572a0056f3, Build 2025-11-05 02:26:37)
Board: promicro_uf2
SOC: nrf52840
Target: promicro_uf2/nrf52840/spi
IMU: ICM-45686
Interface: SPI
Address: 0x00F2
Accelerometer matrix:
0.00000 1.00000 0.00000 0.00000
0.00000 0.00000 1.00000 0.00000
0.00000 0.00000 0.00000 1.00000
Gyroscope bias: 0.00000 0.00000 0.00000
Fusion: VQF
Tracker ID: 1
Device address: A3ABE2433796
Receiver address: EAAF99CF4E82
Battery: 27%
Remaining runtime: 7h 20min
Fully charged runtime: 27h 31min
```

此处我已经配对过, 所以 `Receiver address` 有显示. 这里重点关注 `Device address`.

3. 使用接收器对应的 SmolSlimeConfigurator 实例使接收器进入配对模式, 然后长按 tracker 的按钮至少三秒, 在 tracker 对应的 SmolSlimeConfigurator 实例中应该看到类似如下的输出

```
[20:02:28.176,422] <inf> status: Button pressed
[20:02:28.176,452] <inf> status: Status: 90
[20:02:29.144,592] <inf> system: User shutdown requested
[20:02:32.333,129] <inf> status: Cleared status: 2
[20:02:32.333,160] <inf> status: Status: 88
[20:02:33.145,111] <inf> system: Pairing requested
[20:02:33.155,303] <inf> esb_event: Pairing requested
[20:02:33.155,334] <inf> status: Cleared status: 64
[20:02:33.155,334] <inf> status: Status: 24
[20:02:33.234,619] <inf> esb_event: Pairing
[20:02:33.234,710] <inf> esb_event: Device address: A3ABE2433796
[20:02:33.234,710] <inf> esb_event: Checksum: 72
[20:02:34.235,565] <inf> esb_event: Paired
```

这里看到 `Paired` 即可松手.

4. 在接收器对应的 SmolSlimeConfigurator 实例点击 “List”, 应该能看到类似如下输出

```
>>> list
list
Stored devices:
90855A7370F9
A3ABE2433796
25B9BCBE4551
E6A5C759DA8A
0E6389D6236D
0B4BF4FE7330
50C39218E879
D1F1D0DCFB8C
```

```
A56ECB442ECD
4E1DFDF79F16
```

如果能在这里找到先前看到的 tracker 地址, 则配对完成. 由于我已经把十个 tracker 都配对完成, 所以这里全都显示了

5. 断开 tracker 与电脑的连接

5. 重复上述步骤, 配对所有 tracker

4. 检查 tracker 是否正常工作
   1. 安装 SlimeVR Server
   2. 打开 SlimeVR Server, 按步骤完成设置, 有些与佩戴相关的步骤可以直接跳过
   3. 在 SlimeVR Server 的 Home 界面能看到所有 tracker. 当 tracker 剧烈移动的时候, 这个 tracker 对应的图标会亮起. 点进每个 tracker 可以看到该 tracker 的 3D 预览画面, 在现实中移动 tracker, 检查预览画面的移动是否匹配.

1.3.4 组装 tracker 与外壳, 绑带

我的外壳参考的是 tbd 设计的 tbd, 绑带参考的是 Depact 设计的 Depact V2 Smol 绑带.

1. 下载外壳的 3D 模型文件, 这里我们需要的是 `tbd`, `tbd`, `tbd` 三个模型, 每个模型打印十个即可.
2. tbd

1.3.5 调试, 测试

tbd

1.4 总结

tbd

2025-11-03

Udon Script 分析

对 Udon Script 编译产物 Udon Program 和 Udon VM 虚拟机的分析.

我想知道一些 VRChat 地图的脚本逻辑, 但是 VRChat 地图的脚本都被编译成了一些神秘的, 无法被 AssetRipper 轻易解析的 MonoBehaviour.

我不太了解 VRChat 的世界创作生态, 但是朋友告诉我这是 Udon Script, 还告诉我他也没法解读这些产物.

既然如此, Challenge Accepted!

2.1 Udon Program

Udon Script 的编译产物.

2.1.1 资产

使用 AssetRipper 解包地图后, 能得到大量 AssetRipper 无法正确解析的 MonoBehaviour 资产文件. 其中一些 MonoBehaviour 资产包含一个很长的 `serializedProgramCompressedBytes`. 这代表这个资产是一个 Udon Script 的编译产物.

`serializedProgramCompressedBytes` 是一个十六进制字符串, 是 GZip 压缩后的 Udon Program 序列化结果.

2.1.2 Udon Program 的反序列化

serializedProgramCompressedBytes 经过 GZip 解压后得到的二进制文件是 `UdonProgram` 实例序列化后的结果.

这个序列化过程使用的是一个 VRChat 修改的 `OdinSerializer`. 所以我们可以直接用这个序列化器对应的反序列化器进行反序列化. 一些关键代码如下

```
using System.IO;
using VRC.Udon.Common;
using VRC.Udon.Serialization.OdinSerializer;

using var memoryStream = new MemoryStream(fileData);
var context = new DeserializationContext();
var reader = new BinaryDataReader(memoryStream, context);
UdonProgram program =
  VRC.Udon.Serialization.OdinSerializer.SerializationUtility
    .DeserializeValue<UdonProgram>(reader);
```

2.1.3 `UdonProgram` 类

`UdonProgram` 类中几乎有我们需要的一切. 下面是一个简化[1]的类定义

[1]本小节中出现的类定义只列出了进入序列化后的 Udon Program 二进制的部分.

```
public class UdonProgram : IUdonProgram
{
  public string InstructionSetIdentifier { get; }
  public int InstructionSetVersion { get; }
  public byte[] ByteCode { get; }
  public IUdonHeap Heap { get; }
  public IUdonSymbolTable EntryPoints { get; }
  public IUdonSymbolTable SymbolTable { get; }
  public IUdonSyncMetadataTable SyncMetadataTable { get; }
  public int UpdateOrder { get; }
}
```

我们比较关心 `ByteCode`, `Heap`, `EntryPoints`, `SymbolTable` 这几个字段.

2.1.3.1 Udon 字节码和指令集

是一系列大端序 `u32` 组成的指令的序列.

指令格式为 `OPCODE[OPERAND]`, 两部分各 4 字节, `OPERAND` 是一个大端序 `u32`.

`OPCODE` 包括无参数的 `NOP`, `POP`, `COPY` 和有一个参数的 `PUSH`, `JUMP_IF_FALSE`, `JUMP`, `EXTERN`, `ANNOTATION`, `JUMP_INDIRECT`.

各 `OPCODE` 对应的值为

```
class OpCode(IntEnum):
    NOP = 0
    PUSH = 1
    POP = 2
    JUMP_IF_FALSE = 4
    JUMP = 5
    EXTERN = 6
    ANNOTATION = 7
    JUMP_INDIRECT = 8
    COPY = 9
```

各 OPCODE 和 OPERAND 含义如下:

- `NOP`: 空指令
- `PUSH I`: 将立即数 `I` 压栈
- `POP`: 从栈中弹出一个值并丢弃
- `COPY`: 复制堆中的值
- `JUMP_IF_FALSE ADDR`: 条件跳转到 `ADDR`
- `JUMP ADDR`: 无条件跳转到 `ADDR`
- `EXTERN F`: 调用外部函数, `F` 是堆中的函数签名 `string` 或者函数委托 `UdonExternDelegate` 的地址
- `ANNOTATION`: 注解, 执行时跳过
- `JUMP_INDIRECT IADDR`: 间接跳转到 `IADDR` 作为堆地址指向的值

2.1.3.2 堆

用于存储 Udon VM 执行该 Udon Program 时堆的初始值, 相当于常量段.

简化的类定义如下

```
[Serializable]
public sealed class UdonHeap : IUdonHeap, ISerializable
{
```

```
  [NonSerialized]
  private readonly IStrongBox[] _heap;
  [NonSerialized]
  private readonly Dictionary<Type, Type>
    _strongBoxOfTypeCache = new Dictionary<Type, Type>();
  [NonSerialized]
  private readonly Dictionary<Type, Type>
    _strongBoxOfTContainedTypeCache = new Dictionary<Type, Type>();

  public void GetObjectData(
    SerializationInfo info, StreamingContext context
  )
  {
    List<ValueTuple<uint, IStrongBox, Type>> list =
      new List<ValueTuple<uint, IStrongBox, Type>>();
    this.DumpHeapObjects(list);
    info.AddValue("HeapCapacity", Math.Max(0, this._heap.Length));
    info.AddValue("HeapDump", list);
  }

  public void DumpHeapObjects(
    List<ValueTuple<uint, IStrongBox, Type>> destination
  )
  {
    uint num = 0;
    while (num < this._heap.Length)
    {
      IStrongBox strongBox = this._heap[num];
      if (strongBox != null)
      {
        destination.Add(new ValueTuple<uint, IStrongBox, Type>(
          num,
          strongBox,
          strongBox.GetType().GenericTypeArguments[0]
        ));
      }
      num += 1;
    }
  }
}
```

我们感兴趣的就是其中的 `HeapDump`, 这是一个 `(Addr, Value, Type)` 三元组的列表.

2.1.3.3 入口点表

实际上是函数表.

简化的类定义如下

```
[Serializable]
public sealed class UdonSymbolTable : IUdonSymbolTable, ISerializable
{
  private readonly ImmutableArray<string> _exportedSymbols;
  private readonly ImmutableDictionary<string, IUdonSymbol> _nameToSymbol;

  void ISerializable.GetObjectData(
    SerializationInfo info, StreamingContext context
```

```
  )
  {
    info.AddValue(
      "Symbols",
      this._nameToSymbol.Values.ToList<IUdonSymbol>()
    );
    info.AddValue(
      "ExportedSymbols",
      this._exportedSymbols.ToList<string>()
    );
  }
}

[Serializable]
public sealed class UdonSymbol : IUdonSymbol, ISerializable
{
  public string Name { get; }
  public Type Type { get; }
  public uint Address { get; }

  void ISerializable.GetObjectData(
    SerializationInfo info, StreamingContext context
  )
  {
    info.AddValue("Name", this.Name);
    info.AddValue("Type", this.Type);
    info.AddValue("Address", this.Address);
  }
}
```

这里每个 `UdonSymbol` 里的

- `Name` 是函数名
- `Address` 是该函数的首条指令在 `UdonProgram.ByteCode` 中的索引
- `Type` 无意义

这给我们带来了很多方便.

2.1.3.4 符号表

类定义和入口点表相同, 其中每个 `UdonSymbol` 里的

- `Name` 是符号名
- `Address` 是该符号在堆中的地址
- `Type` 是符号类型

2.2 Udon VM

是一个简单的栈式虚拟机.

2.2.1 堆, 栈和寄存器

- 堆: 是一个 `IStrongBox[]`, 地址就是数组索引, 使用程序中的常量段初始化
- 栈: 一个 `u32` 栈
- PC: 单位是字节

2.2.2 外部函数

Udon VM 的外部函数委托是 `UdonExternDelegate`, 具体定义为

```
delegate void UdonExternDelegate(IUdonHeap heap, Span<uint> parameterAddresses);
```

也即传入
- 堆用于获取参数和写入结果
- 一系列参数地址(在堆中的)用于获取参数

在此基础上封装了 CachedUdonExternDelegate, 具体定义为

```
class CachedUdonExternDelegate
{
  public readonly string externSignature;
  public readonly UdonExternDelegate externDelegate;
  public readonly int parameterCount;
}
```

CachedUdonExternDelegate 可以完全通过一个 string 获取, 也即 externSignature.

这个 externSignature 其实就是简单的函数签名, 如

```
ExternVRCEconomyIProduct.__Equals__VRCEconomyIProduct__SystemBoolean
ExternVRCEconomyIProduct.__get_Buyer__VRCSDKBaseVRCPlayerApi
ExternVRCEconomyIProduct.__get_Description__SystemString
ExternVRCEconomyIProduct.__get_ID__SystemString
ExternVRCEconomyIProduct.__get_Name__SystemString
```

这个签名由两部分组成, 分别是 ModuleName 和 FuncSignature. 类(也即 Module)通过实现 IUdonWrapperModule, 将自己的 ModuleName 和所有 FuncSignature 及其对应的参数数量注册到 UdonWrapper 中, 供其使用完整的 externSignature 获取.

2.2.3 执行过程

读取当前 PC 处的指令
- NOP: PC 步进 4 字节
- PUSH: 把 OPERAND 作为立即数压栈, PC 步进 8 字节
- POP: 弹栈, 丢弃栈顶值, PC 步进 4 字节
- JUMP_IF_FALSE: 栈顶是堆地址, 弹栈, 读该地址对应的堆元素(bool)的值
  - 若为 true, PC 步进 8 字节
  - 若为 false, 设置 PC 为 OPERAND
- JUMP: 设置 PC 为 OPERAND
- EXTERN: 调用外部函数. 尝试读取 OPERAND 作为堆地址指向的对象
  - 若为 string, 通过 UdonWrapper 获取该 string 对应的 CachedUdonExternDelegate
  - 若为 CachedUdonExternDelegate, 也得到了 CachedUdonExternDelegate

  从栈中连续弹出 CachedUdonExternDelegate.parameterCount 个参数地址, 按与弹栈相反的顺序(也即最初的栈顶为最后一个地址)组装成 Span<uint> parameterAddresses, 并调用 UdonExternDelegate. PC 步进 8 字节
- ANNOTATION: PC 步进 8 字节
- JUMP_INDIRECT: 设置 PC 为 OPERAND 作为堆地址指向的 u32 值
- COPY: 从栈中先后弹出 TARGET 和 SOURCE 两个地址, 然后把堆中 TARGET 地址指向的值使用 SOURCE 地址指向的值覆盖. 所在 PC 步进 4 字节

2.3 反编译

用 Claude 写了一个反编译器. 比之前的反汇编器和反编译器效果好一些, 但是能做的还有很多.

2025-06-25

计算机网络复习笔记

计算机网络复习笔记

目录

3.1. 绪论

将自治的计算机相互连接在一起的系统称为计算机网络. 通信子网中只有两种元素: 网络节点和通信链路.

按照地理范围递增, 网络包括

- PAN 个域网(Personal)

- LAN 局域网(Local)
- MAN 城域网(Metro)
- WAN 广域网(Wide)

单位:
- 字节: 大 B, 存储介质中存储的数据量
- 比特: 小 b, 吞吐量

参考模型: OSI 七层, TCP/IP 四层

OSI 模型	TCP/IP 模型	功能
应用层	应用层	应用
表示层		数据表示, 格式转换, 加解密
会话层		管理两个端点的通信, 建立, 维护, 拆除会话
传输层	传输层	端点(不一定是主机)间的数据段传输
网络层	网络层	路由
链路层	网络接口层	可靠帧传输
物理层		透明比特流传输

对等实体之间的通信经过了 U 形的协议栈处理, 是虚拟通信. 而 U 型通道中相邻的通信是实通信.

ICANN 管理 IP 地址资源分配和顶级域名, 前身是 IANA.

ISOC 是互联网协会, 下辖
- IAB: 互联网结构委员会
- IETF: 制定互联网标准(STD 文档, RFC(Request for Comment)文档)的, 解决实际的**工程**问题
- IRTF: 互联网技术**研究**
- 等

认为 IETF 和 IRTF 是 IAB 的下属.

服务和协议是完全相分离的, 于是能自由更改协议而不影响提供的服务.

IEEE 802 包括
- 802.1: 网络互连
  - 802.1Q: VLAN
  - 802.1X: 基于端口的网络访问控制
  - 802.1D: 生成树协议 STP
- 802.2: 逻辑链路控制 LLC
- 802.3: 以太网
- 802.11: 无线局域网 WiFi

3.2. 物理层

透明传输比特流.

规定了:
- 机械特性: 接口所用接插件的形状, 尺寸, 引脚数目, 排列方式
- 电气特性: 接口的各条线上应有的电压范围
- 功能特性: 出现某种电平时的意义, 如高电压代表 1
- 过程特性: 实现特定功能的事件应该出现的顺序

信号传输有同步和异步, 主要看两边有没有一样的时钟. 同步的特点是:
- 收发方使用严格一致的时钟
- 比特传输效率高
- 实现复杂

例 3.1

在一根有传输延迟为 5ms 的 4Mbps 链路上发送 500 字节的消息, 此消息从发送到传播至目的地的延迟共有多少?

解

链路的吞吐量(带宽)决定了发送数据的速度, 延迟决定了最后一个数据包从发出到到达的时延, 于是

$$L = \frac{M}{R} + D = \frac{500\ \text{Bytes}}{4\ \text{Mbps}} + 5\ \text{ms} = 6\ \text{ms}$$

例 3.2

在一个传播延迟为 4ms 的 5Mbps 互联网访问链路上, 传输数据最大数量是多少?

解

这里的“传播延迟”其实是“可以容忍的从首个数据发出到最后一个数据接收之间的时间”, 所以

$$\text{BD} = RD = 5\ \text{Mbps} \cdot 4\ \text{ms} = 2500\ \text{Bytes}$$

例 3.3

1bit 在一个传输速度为 1 Gbps 的有线网络上可以传播多远? 假设通过线传播的信号的传播速度是真空中光速的 2/3.

解

其实是想问“最后一个数据发出时, 第一个发出的数据已经传播了多远”, 于是

$$\frac{1\ \text{bit}}{1\ \text{Gbps}} \cdot \frac{2}{3} \cdot c = 20\ \text{cm}$$

3.2.1. 信道的最大数字带宽

数字带宽是理想的, 静态的, 而吞吐量是实际可测得的.

定理 3.2.1 (奈奎斯特定理)

显然超过 $S_{\max}$ 的采样率没有意义, 所以没有噪声的理想信道的数字带宽上限是

$$R_{\max} = S_{\max} \cdot \log_2 L$$

其中

$$S_{\max} = 2 \times B$$

是最大采样率, 是每秒采样的次数, B 是频带范围, L 是信号的离散级别, 比如采样两个二进制位, 则 $L = 2^2 = 4$.

定理 3.2.2 (香农定理)

在有噪声的信道里, 有

$$R_{\max} = B \times \log_2(1 + S/N)$$

其中 S/N 是信噪比, 当其单位为分贝时, 有

$$S/N(\text{db}) = 10 \times \log_{10} S/N$$

例 3.4

有一条 4 kHz 的无噪声信道, 每秒采样 8000 次, 如果每个采样是 16 比特, 则信道的最大传输速率是?

解

其实我们不关心他实际上每秒采样多少次, 只关心信道本身的性质.

$$R_{\max} = 2 \cdot 4000 \cdot 16 = 128 \text{ kbps}$$

例 3.5

如果一个二进制信号通过一条 4 kHz 的噪声信道, 噪声是 30 分贝, 则最大传输速率是?

解

$$R_{\max} = 4000 \cdot \log_2(1 + 1000) \approx 40 \text{ kbps}$$

例 3.6

如果一条信道的带宽在 3MHz 和 4MHz 之间, 且信噪比是 24 分贝, 问:
1. 信道的传输能力(最大传输速度)如何?
2. 为了达到这个传输能力, 信号级别需要多少级?

解

注意两个公式里面的 B 都是频带范围, 所以这里的 B 应该是

$$B = 4 \text{ MHz} - 3 \text{ MHz} = 1 \text{ MHz}$$

1. $$S/N = 10^{\frac{24}{10}}$$
$$R_{\max} = B\log_2(1+S/N) \approx 8 \text{ Mbps}$$
2. $$R_{\max} < 2B\log_2 L \Rightarrow L \geq 16$$

3.2.2. 编码和调制

- 编码: 对数字信号
  - 归零 RZ: 正负电平, 发送完回到 0
  - 非归零 NRZ: 正电平和负电平
  - 非归零逆转 NRZI: 传输 0 时逆转信号, 传输 1 时不变
  - 曼彻斯特: 使用向上, 向下跳变行为编码 0, 1
  - 差分曼彻斯特: 在时钟开始的时候使用跳变和不跳变编码 1 和 0, 在每个周期中间都跳变用于同步时钟.
  - xb/yb 编码: 用 y 位前缀码编码 x 位数据.
- 调制: 对模拟信号
  - 调幅
  - 调频
  - 调相
  - 综合调制: QAM-n 代表有 n 个信号级别, 每个码元表示 $\log_2 n$ 比特数据

信号经过傅里叶分析可以分解成不同频率的谐波, 谐波的频率是基频的整数倍.

3.2.3. 复用

在单个信道中传输多个信号.

- 时分复用 TDM: 在小的时间片上轮流使用信道. 一种方法是均分, 还可以使用统计时分复用(STDM)的方法根据用户需求分配时间片.
- 频分复用 FDM: 把物理带宽分为若干频率范围, 形成子频带和子信道, 子信道之间可以有很窄的保护带. 计算子信道带宽时, 直接用总带宽除以子信道数量. 有一种技术是正交频分复用(OFDM), 相邻信号必须正交, 但是信道可以重叠, 从而能提升信道的总带宽.
- 波分复用 WDM: 光信号的 FDM.
- 码分复用 CDM: 把每个连接的一个比特分为到 m 个时间间隔中发送, 这些离散的时间是**码片**, 把 1 编码为 m 维的双极向量(分量为 ± 1), 称为**码片序列**. 所有用户的码片序列两两正交, 所以可以全都混在一起线性叠加地发送, 接收方只需要用**发送方**的码片和接收的叠加信号点积, 观察结果 $(m, 0, -m)$ 就能得知发送内容.

例 3.7

五个用户使用 TDM 或 FDM 共享 1 Mbps 链路. 使用 TDM 的每个用户都要以一个固定的顺序轮流完全占据链接 1 ms. 使用 FDM 的每个用户在所有时间中获得 1/5 的链路. 当用户传输一个 1250 字节的消息时, 哪个方法具有最低的可能延迟, 且该延迟时间是多少?

解

- TDM: 每个 1ms 内可以发送
$$1 \text{ ms} \cdot 1 \text{ Mbps} = 125 \text{ Bytes}$$

故共需要等待 9 个 4 ms 和发送 10 个 10 ms, 共 46 ms.

- FDM:

$$\frac{1250\ \text{Bytes}}{\frac{1\ \text{Mbps}}{5}} = 50\ \text{ms}$$

故为 TDM, 46 ms.

3.2.4. 介质

- 同轴电缆
- 双绞线

	屏蔽双绞线 STP	非屏蔽双绞线 UTP
外屏蔽层	有	有
线对屏蔽层	有	无
尺寸	大	小
重量	重	轻
安装	难	易
特点		局域网中广泛使用

- 光纤: 光纤中不同入射角的光线可以互不干扰地传播, 每个入射角是一个“模式”, 因此有多模光纤和单模光纤. SONET 是光纤标准(O 是 Optical).

	单模光纤	多模光纤
发光器件	半导体激光器	发光二极管
发光器件价格	贵	便宜
发光器件寿命	短	长
传输距离	长	短
带宽	大	小
芯线	细	粗
规格	内 8-10 微米, 外 125 微米	62.5 / 125 微米

  模式数量 → 内径(模式多的需要粗) → 传输距离(细长短粗) → 激光器价格(传输距离长的贵) → 激光器寿命(贵的寿命短).
- 非导引性介质, 如微波. 微波容易发生多径衰落.

有线介质传播距离: **光纤** > **同轴电缆** > **屏蔽双绞线** > **非屏蔽双绞线**.

3.2.5. PSTN

由交换局, 本地回路, 干线组成. 使用 PCM(脉冲编码调制技术) 把本地回路上的模拟信号数字化.

一些干线规格:

等级	速率 (Mbps)	路数	说明
Ex 系列(欧洲/中国标准)			

E1	2.048	30	
E2	8.448	120	4 倍 E1
E3	34.368	480	4 倍 E2
E4	139.264	1920	4 倍 E3
E5	565.148	7680	4 倍 E4
Tx 系列(北美/日本标准)			
T1	1.544	**24**	
T2	6.312	96	4 倍 T1
T3	44.736	672	7 倍 T2
T4	274.176	4032	6 倍 T3

T1 干线上复用 24 路语音, 每路电话每帧发送 8 位, 其中 7 位是用户数据, 1 位是控制信号, 此外还有额外的 1 位控制开销, 所以利用率为

$$\frac{24 \times 7}{24 \times 8 + 1} \approx 87.05\%$$

3.2.6. 设备

- 放大器: 去噪和放大信号, 多用于总线拓扑
- 集线器: 收到信号广播给其他所有节点, 多用于星形拓扑. 现在已经基本不用了

中继器和集线器能扩大冲突域.

3.3. 链路层

负责把数据打包成帧, 并做纠错和流量控制.

包括 MAC(介质访问控制) 子层和 LLC(逻辑链路控制) 子层.

3.3.1. 成帧

- 字节计数法: 帧头标记这一帧多大(包括帧头), 但是第一帧错了就全完了.
- 带字节填充的字节标记法: 用一个特定的定界符 FLAG(一个特殊字节) 来标记首尾. 包长度只能是字节为单位, 如果包中间也出现了 FLAG 就要用特定的转义符 ESC (也是一个特殊字节)转义, ESC 自身也转义.
  - 帧界: 7E -> 7D 5E
  - 转义: 7D -> 7D 5D
- 带位填充的位标志法: 允许一帧为任意长度. 起始和终止标志是 01111110(中间有 6 个 1), 但是中间如果发送了五个连续的 1, 就会强制发送一个 0, 所以永远都不会重复. 接收方收信的时候碰到五个 1 接一个 0 就会把 0 删掉.
- 物理层编码违例: 如果物理层用跳变编码 0 和 1, 那可以留出来连续高(低)电平定界.

3.3.2. 检错和纠错

数据传输可能出现错误, 主要有突发错误和随机错误.

3.3.2.1. 奇偶校验

通过在数据后 append 一位, 保证整个帧的所有 1 的个数是奇数或者偶数.

能检一位错, 海明距离是偶数.

3.3.2.2. CRC

- 发方
  1. 约定 r 阶多项式 $G = \sum_{i=0}^{r} g_i x^i$, 其中 $g_i \in \{0, 1\}$, 于是可以仅取系数表示为 $r+1$ 位二进制数 G, 高阶对应高位.
  2. 对数据 M, 左移 r 位, 也即 $x^r M$.
  3. 作模二除法 $x^r M / G$ 余数为 c
     - 二进制除法中直接使用异或作为加减法, 不需要试商, “够除”只看位数
  4. 把 r append 上 c, 作为真实发送的数据
- 收方
  1. 对收到的信息作模二除法查看是否能被 G 整除, 若不能则出错, 若能则无错.

例 3.8

用标准 CRC 方法来传输位流 10011101. 生成多项式为 $x^3 + 1$.

1. 试问实际被传输的位串是什么?
2. 假设在传输过程中从左边数第三位变反了. 请说明这个错误可以在接收端能否被检测出来.
3. 给出一个该比特流传输错误的实例, 使得接受方无法检测出该错误.

解

$$G = 1001$$

1. 有

$$
\begin{array}{rcccccccccccc}
 & & & & & & & & & & & & \\
1001 &) & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 1 & & & & & & & \\
\cline{3-6}
 & & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & & & \\
 & & & & & & 1 & 0 & 0 & 1 & & & \\
\cline{5-8}
 & & & & & & 0 & 1 & 0 & 0 & 0 & & \\
 & & & & & & & 1 & 0 & 0 & 1 & & \\
\cline{6-10}
 & & & & & & & 0 & 0 & 0 & 1 & 0 & 0 \\
 & & & & & & & & & & 1 & 0 & 0
\end{array}
$$

故为 10011101100.

1. 若第三位取反, 也即接收到 10111101100, 则

$$
\begin{array}{rcccccccccccc}
 & & & & & & & & & & & & \\
1001 &) & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\
 & & 1 & 0 & 0 & 1 & & & & & & & \\
\cline{3-6}
 & & 0 & 0 & 1 & 0 & 1 & 1 & & & & & \\
 & & & & 1 & 0 & 0 & 1 & & & & & \\
\cline{5-8}
 & & & & 0 & 0 & 1 & 0 & 0 & 1 & & & \\
 & & & & & & 1 & 0 & 0 & 1 & & & \\
\cline{6-10}
 & & & & & & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

余数不为 0, 故检测到出错.

2. 只需要令正确数据再异或(加)上除数即可

$$10011101100 \oplus 1001 = 10011100101$$

3.3.2.3. 互联网校验和

1. 发送方: 把数据按一定比特长度分为多个位串, 然后执行反码求和, 得到的即为校验和
2. 接收方: 把收到的数据按同样方式分为多个位串, 把这些位串和校验和做反码求和, 得到 0 即为无错误, 得到 1 则为有错误.

反码求和也即先求和, 溢出部分当作加数继续求和, 直到没有溢出后, 对结果取反.

3.3.2.4. 纠 1 位错的海明码

两个位串中不同的位的个数为这两个位串的海明距离(也即 xor 后统计 1 的个数).

一个编码中任取两个码字, 最小的海明距离就是这个编码的海明距离.

海明距离为 $d+1$ 的编码方案能实现 d 位的错误检测, 因为只有 $d+1$ 位同时发生错误才有可能不被察觉地从一个编码变为另一个.

令码字长度

$$n = m + r$$

求出最小 r 满足

$$m + r + 1 \leq 2^r$$

这 n 位中 r 位是校验码, 分布在从第一位开始的 2 的幂次位上.

编码时, 把原数据每一位的序号分解为 2 的幂的和. 然后对于每一个 2 的幂位置的校验码, 取出所有分解包含该位置序号的位, 把这些位异或(或者做奇偶校验)就能得到这个校验码的值.

解码时, 按编码的方式重新计算校验位, 并且统计所有不一致的校验位. 如果全都一致, 那么认为没有错误, 如果有不一致, 把所有出错校验位的序号加起来就是出错的实际位的序号, 把那一位取反即可.

例 3.9

偶校验纠 1 位错海明码编码 10101111.

解

$$m = 8$$
$$m + r + 1 \leq 2^r \Rightarrow r \geq 4$$

使用 4 位纠错码, 则

序号	1	2	3	4	5	6	7	8	9	10	11	12
值			1		0	1	0		1	1	1	1
纠错码 1	1		√		√		√		√		√	
纠错码 2		0	√			√	√			√	√	
纠错码 4				0	√	√	√					√
纠错码 8								0	√	√	√	√

结果	1	0	1	0	0	1	0	0	1	1	1	1

例 3.10

偶校验纠 1 位错海明码纠错 100110001100, 其中 $m = 8, r = 4$.

解

序号	1	2	3	4	5	6	7	8	9	10	11	12
值	1	0	0	1	1	0	0	0	1	1	0	0
纠错码 1	0		√		√		√		√		√	
纠错码 2		1	√			√	√			√	√	
纠错码 4				1	√	√	√					√
纠错码 8								0	√	√	√	√
结果	✗	✗		√				√				

故第 $1 + 2 = 3$ 位出错, 取反, 正确值为 101110001100.

3.3.3. 可靠传输

3.3.3.1. 停等

接收方收到正确数据返回一个确认帧 ACK, 否则什么都不发. 发送方收到确认帧就发下一帧, 若长时间没有收到就重传这一帧(ARQ, 自动重复请求协议).

由于是一帧一帧地发, 所以只需要一位二进制标识帧号, 在发送和确认时传送.

信道利用率

$$\text{util} = \frac{T_{\text{data}}}{T_{\text{data}} + T_{\text{ACK}} + 2l}$$

常常认为 T_{ACK} 为 0, 因为比较小. l 为延迟.

例 3.11

使用 ARQ 协议在一个 1 Mbps 链路上发送一系列的 1250 字节消息. 此链路的传播延迟为 5 ms. 问可以使用的链路带宽的最大百分比是多少?

解

$$T_{\text{data}} = \frac{1250 \text{ Bytes}}{1 \text{ Mbps}} = 10 \text{ ms}$$

$$\text{util} = \frac{10 \text{ ms}}{10 \text{ ms} + 2 \cdot 5 \text{ ms}} = 50\%$$

3.3.3.2. 滑动窗口

同时进行 w 个停等.

信道利用率

$$\text{util} = \frac{wT_{\text{data}}}{T_{\text{data}} + T_{\text{ACK}} + 2l}$$

可以看出只要窗口够大, 利用率就能达到 100%.

出错时有两种方案

1. 回退 n 帧 GBN: 从出错那帧(没收到 ACK 开始)全部重发, 需要发送方有较大缓冲区, 因为发送方需要重传. 发送窗口最大为 MAX_SEQ, MAX_SEQ 为最大序列号(从 0 开始), 接收窗口固定为 1. 可以结合无滑动窗口的情况记忆, 也即序号从 0 开始, 最大为 1 , 窗口也为 1 .
2. 选择性重传: 只重传出错的帧, 需要接收方有较大缓冲区, 因为要等待和排序收到的帧. 窗口最大为 $(\text{MAX\_SEQ} + 1)/2$

这种最大窗口区别的本质是回退 n 帧的渐进的, 即使只确认了一个也能向下滑动, 而选择性重传是累计的, 本窗口必须全部完成才能向下滑动. 因此回退 n 帧只需要提供 1 的冗余, 而选择性重传需要提供一个完整窗口的冗余.

选择性重传不需要重传大量帧, 效率比较高.

> **例 3.12**
>
> 两台主机之间的数据链路层采用了回退 n 帧协议(GBN)传输数据, 数据传输速率为 16 kbps, 单向传播延迟是 270 ms, 数据帧长度为 128 - 512B, 接收方总是以数据帧等长的帧进行确认, 为使得信道利用率达到最高, 帧序号的比特数至少为多少位?
>
> > **解**
> >
> > 要求至少, 所以我们求上界, 取数据帧长度 128 Bytes.
> >
> > $$T_{\text{data}} = T_{\text{ACK}} = \frac{128\ \text{Bytes}}{16\ \text{Kbps}} = 64\ \text{ms}$$
> >
> > $$\text{util} = w\frac{64\ \text{ms}}{2 \cdot 64\ \text{ms} + 2 \cdot 270\ \text{ms}} \Rightarrow w > 10$$
> >
> > 取 $w = 11$, 则帧序号至少有 $0 \sim 11$, 也即至少 4 位.

3.3.4. 数据链路控制(LCP)协议

3.3.4.1. HDLC

带位填充的位标志法成帧, CRC 检错.

链路控制功能包括连接的建立, 数据的传输, 连接的断开.

信息帧用来传有效信息和数据.

3.3.4.2. PPP

3.3.4.3. PPPoE

3.4. 介质访问控制和局域网

一条信道需要被许多工作站共享, 因此需要考虑信道的分配和使用, 这就是 MAC 介质访问控制, 这部分功能由 MAC 层完成.

局域网信道包括
- 广播式信道: 所有节点共享一条信道, 只有这种情况才需要解决冲突的介质访问控制方法. 总线拓扑, 星形拓扑, 环形拓扑都属于广播式信道
- 点对点信道: 两个节点专用的通信链路. 如点对点拓扑

3.4.1. 多路访问协议

目的是为了让每台工作站都有信道的部分使用权. 分为
- 随机访问协议
- 受控访问协议
- 有限竞争协议

3.4.1.1. 随机访问协议

工作站不是提前获得预分配的资源, 而是有了要传输的数据帧之后, 用一种机制竞争共享信道的使用权.

3.4.1.1.1. 纯 ALOHA 协议

“任性”, 所有计算机想发就发.
- 如果发送成功, 接收站回一个确认帧, 网络中有一个特殊计算机会把收到的确认帧广播出去, 发送站收到了确认帧就认为发送成功.
- 如果同时有两台计算机在发送信息, 就会发生碰撞, 发送失败, 无法通过 CRC 校验, 于是会遵循二进制指数回退算法进行重传.

定义 3.4.1 (二进制指数回退算法(BEB))

第 k 次传输失败, 从 $[0, 2^k - 1]$ 中随机抽一个时间等待后再重发.

定理 3.4.2

ALOHA 协议的吞吐量为

$$S = Ge^{-2G}$$

其中 G 是每个标准帧时(传输一个数据帧需要的时间)内产生和重传的平均数据帧数.

3.4.1.1.2. 分槽 ALOHA 协议

把时间离散化, 存在某些同步的时隙的开始, 只有在开始处才能发信.

定理 3.4.3

分槽 ALOHA 协议的吞吐量为

$$S = Ge^{-G}$$

3.4.1.1.3. 载波侦听多路访问(CSMA, Carrier Sense Multiple Access)系列

3.4.1.1.3.1. 非持续 CSMA

要发信的时候先侦听信道是否在使用, 如果在用, 就先等待一个随机时间再来, 如果没在用, 就发.

3.4.1.1.3.2. p-持续 CSMA

要发信的时候先侦听信道是否在使用, 如果在用, 那就一直监听, 直到空闲. 如果发现空闲, 有概率 p 发信, $1-p$ 空出一个时隙再监听.

3.4.1.1.3.3. 1-持续 CSMA

p-持续 CSMA 的一种特例.

3.4.1.1.3.4. 带冲突检测的 CSMA (CSMA/CD)

是 1-持续 CSMA 的改进.

即使使用上述策略, 还是有冲突的可能.

CSMA/CD 的网卡有发送器 Tx 和接收器 Rx, Tx 发信的时候也给 Rx 一份, Rx 同时也会监听网络上的信号, 如果 Rx 监听到两个信号一样, 就说明没冲突, 如果检测到冲突, 就停止发送, 并且广播一个拥塞信号 jam.

冲突只能在开始发送后 $2D$ (D 是最远的一台机器的时延)时间内发生, 这是显然的: 极端情况本机消息的头部(已经传播了 D)与最远的一台机器发送的另一条消息的头部发生碰撞, 最远的机器马上广播 jam, 在 D 后该 jam 才被本机接收. 这个时间是冲突窗口, 只要在这个时间内没有检测到冲突, 就说明发送成功.

同时, $2D$ 也是最小帧长, 因为如果比此长度还短, 即使检测到冲突, 消息也已经发完了.

经典以太网使用 CSMA/CD, 除此了上面描述的之外, 还有以下设计
- 检测到冲突后重发的等待时间使用 BEB
- 如果失败次数超过了 15 次, 放弃重发, 所以“每一次冲突都会重传, 都会随机等待两倍时间”是错的

因此以太网的 MAC 协议提供的是无连接不可靠的服务, 因为显然没有连接, 而且可能放弃.

> **例 3.13**
>
> 在经典以太网中, 有 A, B 和 C 共 3 个站点, 采用 CSMA/CD 和 BEB(二进制指数回退算法). 假设所有站点都未开始发送消息. 当下面这些事件按顺序发生之后, 哪些说法是正确的? 你观察这些事件的顺序:
> 1. A 成功发送了一帧
> 2. A 和 B 都发送了帧, 出现冲突
> 3. A 成功地重新发送了一帧
>
> A. A 将重新等待两个时隙
> B. A 和 B 处于不同的工作状态
> C. B 准备好发送下一帧, 不执行 BEB
> D. B 已经等待了至少两个时隙
>
> > **解**
> >
> > AB 冲突后, 他们按照 BEB 将从 $\{0,1\}$ 个时隙中随机选择进行等待, 之后 A 成功发送而 B 暂时没有发送表明 A 选择等待 0 时隙, B 选择等待 1 时隙, 故显然选择 BD.

例 3.14

假定 1km 长的 CSMA/CD 网络的数据率为 1Gb/s. 设信号在网络上的传播速率为 200000km/s. 求能够使用此协议的最短帧长.

解

$$2D = 2 \cdot \frac{1000}{200000 \cdot 10^3} = 10^{-5}$$
$$1\ \text{Gbps} \cdot 10^{-5} = 10^4\ \text{bits} = 1250\ \text{Bytes}$$

3.4.1.1.3.5. 带冲突避免的 CSMA (CSMA/CA)

有确认机制, 是**无线**局域网中最常见的介质访问控制协议, 这是因为无线不能边发边听, 所以不能用 CSMA/CD.

3.4.1.2. 受控访问协议

3.4.1.2.1. 位图协议

N 台机器发消息之前要先经过一个 B 个比特时间的竞争期, 期间每台机器在各自的一个比特时间内发送 1 举手表示自己有数据要发, 竞争期结束之后是传输期, N 台机器根据比特位从低到高各自排队发送.

3.4.1.2.2. 二进制倒计数协议

上一个协议中, 如果参与的计算机太多, 竞争期就会很大, 效率不高.

每个计算机分配一个 $\log_2 N$ 的编号. 竞争期内相应地也有 $\log_2 N$ 个比特时间, 每个比特时间内各自发送自己的对应位, 所有计算机在同一时间内发送的位求或, 如果一台机器发现结果和自己的对应位不一样, 那就退出竞争, 一直这样决出得到发送权的机器.

有优先级, 编号越大优先级越高.

3.4.1.2.3. 令牌传递协议

令牌在环状的网络中轮流.

- 发信: 取得令牌时, 查看令牌中有无数据
  - 无数据: 把数据插入令牌, 令牌成为数据帧. 当令牌下一次到达时, 本机负责把令牌中的数据删除
  - 有数据: 等待令牌下一次到达

  然后把令牌递给下一台机器
- 收信: 取得令牌时, 查看令牌中有无数据, 数据是否发给自己, 如果是, 那么收到了数据, 然后把令牌递给下一台机器

3.4.2. 以太网

从参考模型看, 这部分设计覆盖了物理层和数据链路层, 但是 IEEE 802.3 只覆盖了物理层和 MAC.

3.4.2.1. 分类

3.4.2.1.1. 经典以太网

使用 CSMA/CD, 曼彻斯特编码, 有以下几种传输介质

特点	10Base-5	10Base-2	10Base-T	10Base-F

线缆类型	粗缆	细缆	双绞线	光纤(Fiber)
最大长度	500m	185m	100m	
安装难度	不易安装	比粗缆易安装		
拓扑结构	物理总线拓扑		物理星状拓扑, 逻辑总线拓扑	

线缆名字中, 10 是指速度为 10mbps, 前二者的 5 和 2 指的是传输距离, 后二者的 T 和 F 指的是传输介质.

3.4.2.1.2. 交换式以太网

集线器, 中继器慢慢被交换机替代, 形成了交换式以太网.

交换机可以对所连接的任意两个接口进行无冲突的通信(通过交换机内部 n^2 的阵列交换点形成虚拟电路), 端口和工作站形成无冲突域, 每个端口所在的 LAN 段形成独立冲突域, 效率高.

如果交换机端口不是全双工, 而是半双工, 那还是要用 CSMA/CD 防止冲突.

3.4.2.1.3. 快速以太网

各以太网版本和速度:

名称	速度
经典以太网	10 Mbps
快速以太网	100 Mbps
千兆以太网	1 Gbps
万兆以太网	10 Gbps
其他更快的	顾名思义

3.4.2.2. 帧格式

精品背书课.

3.4.2.2.1. MAC 地址

最高字节的最低位是单播(0)/组播(1)切换位, 次位是全局(0)/本地(1)切换位.

MAC 地址的高 24 位是所谓 OUI 地址, 也即组织唯一标识符, 后 24 位是组织内给设备的标识符.

3.4.2.3. L2 交换

交换机需要按照地址把数据发送到对应端口, 但是交换机是即插即用的透明设备, 需要学习才能知道什么端口是谁.

当数据来到交换机的时候, 以下情况:

1. 交换机的地址表中不存在目标地址, 或者目标地址就是广播地址, 于是广播(向除了源端口的其他端口发送), 并且把源端口和其地址记录(或更新)在路由表中
2. 交换机的地址表中存在目标地址
    1. 目标地址对应的端口就是源端口, 丢弃这一帧, 并更新路由表中的源端口地址
    2. 目标地址对应的端口不是源端口, 把一帧传送到目标端口, 并更新路由表中的源端口地址

表中存储的地址-端口对都是有时限的, 如果长时间没有更新就会删除.

交换方式有三种:

- 存储转发: 缓存完整帧, 然后转发. 转发前要进行 CRC 校验. 出错少, 但是慢
- 直通交换: 读取到帧的目标地址后, 立即在源端口和目标端口之间建立通道, 转发这一帧, 出错多, 但是快

- 无分片交换: 介于两者之间. 接收到帧的前 64 帧再开始转发, 因为冲突往往在刚开始传输的 64 字节内, 这样做可以过滤冲突碎片.

交换机工作在物理层和链路层, 因为他们接收转发物理信号, 而且需要根据物理地址决策.

交换机是全双工, 所以计算全双工状态的总带宽时要注意乘 2.

交换机是工作在混杂模式的设备, 也即网卡会接收所有监听到的包, 无论目标是不是自己, 因为交换机要进行转发, 所以这是必须的. 而正常模式(一般情况下的模式)下, 网卡则只处理目标是自己的包.

网桥就是旧版本的交换机, 行为和交换机一样.

3.4.3. VLAN

人工限制交换机的广播域(分隔广播域), 使用的标准是 IEEE 802.1Q.

3.4.4. 生成树协议 STP

在交换机上运行, 通过选举根节点和到根节点的端口并屏蔽其他端口阻断物理链路中的回路, 解决广播风暴等问题. 使用的标准是 IEEE 802.1D.

3.4.5. 无线技术

- ZigBee: 物联网用的多
- 蓝牙: 基本单位是微微网(Piconet)
- WiFi

只有 WiFi(IEEE 802.11) 是无线局域网协议, 其他都是个域网.

802.11 的帧格式见 各协议 PDU 及相关数值.

其中标志位后八位有一些标志, 如
- 去往 DS(Distributed System, 分发系统): 也即去往 AP
- 来自 DS: 也即来自 AP

四个地址字段的含义也根据这两个标志位的情况改变, 具体来说, 可以是
- DA: 目标地址
- SA: 源地址
- BSSID
- TA: 发送数据帧的站点地址
- RA: 接收数据帧的站点地址

3.5. 网络层

寻路.

提供的服务有两种.

服务	数据报网络	虚电路网络
有无连接	无	有
数据寻径	每个分组独立寻径	每个分组只带有连接号, 不需要自己寻径
抗毁性	故障不会瘫痪网络, 分组可以自由寻路	连接发生故障需要重新搭建
到达顺序	乱序到达	按序到达
状态	无状态	有状态

服务质量	难以保证	可以保证

数据报网络提供的是数据包交换(分组交换, 包交换), 乱序到达是其特点.

3.5.1. IPv4 协议

注意, IPv4 并不能保证分组不丢失.

3.5.1.1. 分组格式

精品背书课.

3.5.1.2. 分片

链路层的 MTU 的限制会导致分片.

IPv4 分组中存在标记位:

- DF(Don't Fragment): 若为 0 表示可以分片, 为 1 表示不要分片
- MF(More Fragment): 若为 0 表示这是最后一个分片, 为 1 表示还有更多分片

如果分组长度 L 大于转出网络 MTU M(对于以太网来说, MTU 是 1500 Bytes), 且分组中 DF 为 0, 那么一片的载荷长度为

$$d = \left\lfloor \frac{M-H}{8} \right\rfloor * 8$$

这里是为了保证分片长度为 8 字节整数倍, 因为分片偏移的单位是 8 字节.

总分片数为

$$n = \left\lceil \frac{L-H}{d} \right\rceil$$

其中 H 为头部长度, 往往为 20.

例 3.15

节点 A 到 B 通过路由器 R1 和 R2 路由. 通过网络发送的 IP 数据报有 20 字节长的报头. A-R1 链路的 MTU 是 1800 字节, R1-R2 链路的 MTU 是 1200 字节, R2-B 链路的 MTU 是 600 字节. 如果 A 想要发送长度为 2800 的消息, 问: B 接收的数据报的总个数是多少? (假定源数据报中的 DF=0).

解

首先计算出各个链路允许的最大有效载荷.

$$\left\lfloor \frac{1800-20}{8} \right\rfloor \cdot 8 = 1776 \text{ Bytes}$$

$$\left\lfloor \frac{1200-20}{8} \right\rfloor \cdot 8 = 1176 \text{ Bytes}$$

$$\left\lfloor \frac{600-20}{8} \right\rfloor \cdot 8 = 576 \text{ Bytes}$$

于是

$$(2800)\begin{cases}1796(1776)\begin{cases}1196(1176)\begin{cases}596(576)\\596(576)\\44(24)\end{cases}\\620(600)\begin{cases}596(576)\\44(24)\end{cases}\end{cases}\\1044(1024)\begin{cases}1044(1024)\begin{cases}596(576)\\468(448)\end{cases}\end{cases}\end{cases}$$

共 7 个.

3.5.1.3. IPv4 地址

本质属性是位置相关性.

一些特殊地址

名称	地址	备注
受限广播地址	全 1	只做目的地址, 向全网主机广播, 实际上只是本地广播
未指定地址	全 0	只用作特殊情况下的源地址
网络广播地址	网络号.全 1	只做目的地址, 向对应网络
网络地址	网络号.全 0	不做源或目的地址, 代表网络本身
网络特定主机地址	全 0.主机号	用来给本网络的某主机发信, 只做目标地址
环回地址	127.*	用于本机测试, 注意 127.0.0.0 不行, 因为是网络地址. 此外, 环回地址不会发送到网络上, 直接在本机处理, 所以不做源地址

私人地址空间:

- 10.*
- 172.16.* - 172.31.*
- 192.168.*

ABCD 类地址: 看第一个八位组开头有几个连续 1, 0 个是 A, 1 个是 B, 依此类推. 其中 D 类是组播地址, E 类保留未用.

类 ABCD 地址: 只看子网掩码有多少网络位

3.5.1.4. IPv4 地址的获取

动态主机配置协议 DHCP.

获取地址有以下状态:

- 初始状态 INIT: 主机开机后进入初始状态, 广播 Discover 消息, 寻找 DHCP 服务器
- 选择状态 SELECTING: 收到 Discover 的服务器单播 Offer 应答消息给主机, 由于局域网内可能不止一台 DHCP 服务器, 所以还需要等待主机的应答, 主机选择一个 Offer 回复 Request 消息
- 请求状态 REQUESTING: 被请求服务器回复 ACK 消息
- 绑定状态 BOUND: 主机收到 ACK 消息后, 正式绑定 DHCP 服务器提供的 IP 地址, 子网掩码, 默认网关, 租期. 租期消耗一半的时候, 再次发送 Request 消息更新租期. 客户端也可以发送 RELEASE 消息主动取消租约
- 更新状态 RENEWING: 发送更新后, 等待 ACK 消息的时期
- 再绑定状态 REBINDING: 更新成功, 租期重置

3.5.1.5. CIDR 无类域间路由

CIDR 是一种路由汇聚技术.

没有 ABCD 类地址限制, 可以按需分配网络号长度(VLSM 可变长子网掩码), 所以
- 缓解 IP 地址耗尽(通过 VLSM)
- 减小路由表规模(CIDR 路由汇聚)
- 按需分配 IP 地址
- 统一管理不同类别 IP 地址

3.5.1.5.1. 子网划分

先分配地址多的, 再分配地址少的, 分配的时候可以基本按照树状的分配方式.

需要注意, 如果子网内有 n 台客户机需要上网, 则实际上至少需要 $n+3$ 个 IP, 因为需要
- 子网本身的地址
- 子网广播地址
- 网关自身的地址

3.5.1.6. 其他 IP 协议或技术

3.5.1.6.1. ARP 地址解析协议

主机发信需要对方的 MAC, 但是有时候只知道 IP 不知道 MAC, ARP 就是用来通过 IP 获得 MAC 的协议.

3.5.1.6.1.1. 数据帧格式

精品背书课.

3.5.1.6.1.2. 基本工作原理

基本工作方式为:
1. 一方发送 ARP 请求, 其中目标硬件地址保留全零
2. 由于目标地址是**广播**地址, 所有主机都会接受
3. 除了目的主机, 其他主机不理睬
4. 目的主机收到后发送 ARP 应答, 其中所有地址都填好, 因此**不再是广播**
5. 源主机收到应答并获得地址

每个主机都会维护一个 ARP 表, 按如下方式更新
- 发出请求, 接到应答后按应答更新
- 收到请求的主机, 无论是不是目标, 都提取请求中的发送方地址更新

实际上记录都有生存时间, 时间太长自动删除. 如果主机启动或者重新配置网卡, 主机会广播一个免费 ARP 请求, 目标和发送方都是他自己, 之前存储过这台机器的接收到这个请求就会用里面的信息更新. 除此之外, 免费 ARP 请求的发送和接收方都是自己, 不期望得到任何回答, 如果得到了, 说明有地址冲突, 所以也可以用来**检测 IP 地址冲突**.

如果是对远程网络中的主机 ARP, 则最开始要把路由器 MAC 设置为目标 MAC, 路由器拿到 ARP 请求后再根据 IP 地址在网络层进一步处理, 直到路由器拿到 MAC 地址, 再重新封装一个 ARP 应答发给源主机.

3.5.1.6.2. ICMP

消息格式中包含类型 1B, 代码 1B, 校验和 2B:
- 差错报告
  - 类型 3
    - 主机不可达消息

  - 端口不可达消息
  - 需要分片但不允许消息
- 类型 11
  - 超时消息
  - 重组超时消息
- 查询信息类型
  - 类型 8: 回声请求
  - 类型 0: 回声应答

几种情况对应的差错:
- 不可达(TTL 耗尽): 超时
- 拥塞: 源抑制

PMTU 利用 ICMP 发现链路上的最大 MTU.

3.5.1.6.3. NAT 网络地址转换

让多个私有 IP 的主机通过同一个公有 IP 连接外部网络, 具体做法就是
1. 内网分组到达 NAT 转换器, NAT 转换器修改源 IP 为公有 IP, 并且分配一个端口号, 对转换前后的端口号和源 IP 做好记录
2. 转发
3. 收到回信, 按照记录好的端口号, 源 IP 修改回信
4. 转发

这个服务是完全透明的.

3.5.2. IPv6 协议

3.5.2.1. IPv6 地址

128 位, 记作 $8 \times 4 \times 4$ 的冒分十六进制. 也有和 IPv4 一样的前缀表示法.

一些特殊地址

名称	地址	备注
站点本地地址	fec0::/48	
组播地址	ff00::/8	
链路本地地址 LLA	fe80::/10	
全球单播地址 GUA	现在往往是 2 或者 3 开头	
子网路由器任播地址	网络号后全 0	

任播地址做目的地址时, 包会发送到最近的一个该地址的接口. 其实任何一个单播地址都能成为任播地址: 只要他被分配给多个接口, 而且这个接口明确知道自己这个地址是任播地址.

省略规则:
- 前导零必须省略, 如 $0001 \to 1$
- 最长零省略, 如果有连续多组四位十六进制全为 0, 那就全都省略为一个 `::`, 最长的优先, 等长最前面的优先, 不省略一组 0.

链路本地地址可以使用 EUI-64 根据 MAC 地址生成, 具体来说
1. 原 MAC 地址为 `AA-BB-CC-DD-EE-FF`
2. 在正中间插入 `FFFE`: `AA-BB-CC-FF-FE-DD-EE-FF`
3. 翻转 U/L 位(最高一组的次低位): `A8-BB-CC-FF-FE-DD-EE-FF`

3.5.2.2. IPv6 过渡技术

- 双协议栈技术
  - 一般: 一个机器同时支持两种协议, 要和什么协议的机器交互就用什么协议
  - 双协议栈过渡机制: 在最后的过渡期服务少数的 IPv4 主机, 在一个纯 IPv6 网络内提供一个类似 NAT 服务的 DSTM 服务器和一个双栈的 TEP 端点, 网络内主机通过 DSTM 获取 NAT 后的 IPv4 地址, 把整个包封在 IPv6 帧里面通过 TEP 和纯 IPv4 主机交互.
- 隧道技术
  - IPv6 over IPv4: 多为协议 41 封装, 也即 IPv4 分组中的协议字段为 41, 数据段即为纯 IPv6 分组
  - IPv4 over IPv6
- 网络地址转换-协议转换 NAT-PT: 通过一个双栈网关作为代理沟通两种纯 vX 的主机(通过做 NAT 和 PT), 但是对于部分双栈机器可以直通. 主要用来转换协议

3.5.2.3. 其他 IPv6 技术

3.5.2.3.1. 邻居发现(ND)

是通过 ICMPv6 实现的. 功能有

- 自动地址配置
- 重复地址检测
- 邻居不可达性检测

3.5.2.3.2. ICMPv6

“ICMPv6 用于主要用于报告 IPv6 分组的传输问题”.

3.5.3. 路由协议

作用是自动学习网络(网络发现)并维护路由表, 从而达到寻路的目的.

局域网内不需要经过路由器(网关), 消息可以直达目标主机.

3.5.3.1. 距离矢量路由协议(DV)

基本上分为三步

- 维护距离矢量
- 交换距离矢量
- 更新距离矢量

每个路由器维护一个他到其他所有节点的距离的矢量, 与邻居交换信息的时候通过自己到邻居这条路径优化自己的矢量.

RIP 协议就是一个 DV 协议的例子.

RIP 协议默认用跳数量度, 每个 RIP 路由器周期性和邻居交换, 默认 30s 一次, 最大量度为 15 跳, 再高认为不可达, 缺点有

- 收敛慢
- 不能到 15 跳之外
- 路径量度不合理

除此之外, 还有 IGRP, BGP 也是 DV.

3.5.3.2. 链路状态路由协议(LS)

基本运作:

- 发现邻居: 通过 Hello 报文
- 设置链路: 了解自己和链路的状况, 并为此链路设置量度, 代价, 开销
- 构造 LSA(链路状态公告): 一个小地图, 包括**邻居**信息, 到邻居的链路, 链路上的量度

- 分发 LSA: 把 LSA 发给其他所有路由器
- 计算: 构建以自己为根, 到其他所有路由器的最短路径, 以及最短路径生成树

3.5.3.2.1. OSPF 协议

开放, 使用带宽量度, 无类, 收敛快, 无路由环, 有层次性.

运行过程

- 建立全毗邻关系: 相互交换数据, 同步数据库
- 选举 DR 指定路由器 和 BDR 备用路由器: 为了减少同步次数
- 发现路由: 从邻居交换的信息里面获取新的路由
- 计算最佳路由
- 维护路由: 默认 30min 更新一次

用到五种报文:

- Hello: 发现邻居, 建立和维护关系
- DD(Database Description): LSA 摘要
- LSR(Link State Request): 要求详细的 LSA
- LSU(Link State Update): 回应 LSR 或主动发送链路变化
- LSAck(Link State Acknowledgement): 确认收到

全毗邻关系的建立过程和状态迁移:

1. 互发 Hello (Down Init 初始状态), 建立双向双边关系 (Two-way 双向状态)
2. 首次交换 DD, 确立主从关系, 交换初始序列号 (ExStart 准启动状态)
3. 交换全部 DD (Exchange 交换状态)
   - 如果拓扑数据库一致, 直接进入全毗邻状态 (Full adjacency 全毗邻状态)
   - 否则, 通过 LSR, LSU, LSAck 报文交互(Loading 加载状态), 最终达到数据库一致, 建立了全毗邻关系 (Full adjacency 全毗邻状态)

3.5.3.3. BGP

前面的协议都是 IGP, 也即在自治系统(AS)内部运行的路由协议. AS 之间运行的是 BGP, BGP 也是一种 DV.

BGP 报文中包含路径属性, 包括

- 起点
- AS 路径: 记录了完整路径, 可以防止环路
- 下一跳
- 多出口辨识属性
- 本地优先属性

3.5.3.4. IP 组播

IGMP 互联网组管理协议: 在主机和组播路由器之间建立, 维护和拆除组播组成员关系.

- 源树: 组播源为根到组播组成员的树, 每个组播源都会生成一个
- 共享树: 组播源发出的分组先到达一个汇聚点, 然后再从汇聚点转发到组播组成员, 每个组播组只有一个

有一系列组播路由协议:

- 域内
  - 密集模式: 只支持源树
    - DVMRP
    - PIM-DM: 使用泛洪-剪枝的方式找到源树, 也即“推”模型

  - 稀疏模式: 同时支持共享树和源树
    - PIM-SM: 接收者需要显式加入, 也即“拉”模型
    - CBT
  - 链路模式
    - MOSPF
- 域外
  - MSDP(链接 PIM-SM 域)
  - MBGP(BGP 多协议拓展)

组播有一系列优势, 包括
- 降低网络流量
- 降低应用服务器的负担
- 降低干线(而不是骨干)上的流量

3.5.4. QoS

区分服务质量.

3.5.4.1. 漏桶算法

数据先传送到一个缓冲区, 从缓冲区中匀速抽取数据发送, 缓冲区满后超出的数据将会被丢弃.

不支持突发.

3.5.4.2. 令牌桶算法

漏桶算法的反演. 匀速向令牌桶中填充令牌, 令牌桶有一定容量, 多出的令牌将被丢弃, 数据只有从令牌桶中取得令牌才能发送.

支持一定程度的突发.

例 3.16

一个网络节点在网卡前接了一个令牌桶和漏桶, 令牌桶的容量 C 为 10000KB, 令牌产生的速率是 25MBps; 漏桶的容量是 8000KB, 输出速率是 125MB/s.

1. 如果网络节点产生了大量分组, 令牌桶的输出速度达到了 50MB/s, 以这样的速度输出, 最多可以持续多长时间?
2. 如果网络节点某个时刻产生的分组突然增加, 产生了 50MB 的分组, 假如令牌桶已经在空闲时间积攒了满桶的令牌, 发送完 50MB 的全部分组, 需要多长时间? 不计算漏桶处理分组的时间.

解

1. 设能持续时间 t, 则有

$$25t + 10 = 50t \Rightarrow t = 0.4\text{s}$$

2. 设以 125 MBps 的速度输出了 d 大小的数据, 则有

$$\frac{d}{125} \cdot 25 + 10 = d \Rightarrow d = 12.5$$

持续 0.1s.

之后又以 25MBps 的速度输出了 37.5 MB 数据, 持续

$$\frac{37.5}{25} = 1.5\text{s}$$

共 1.6s.

3.5.4.3. 多标签交换 MPLS

面向连接, 转发的时候和标签交换(LS)类似, 只需要检查标签, 替换标签, 转发分组. 运行在 OSI 模型的 “2.5” 层.

可以用专用电路实现, 效率高.

3.6. 传输层

实现进程到进程的通信.

有无连接, 区别在于有无虚电路.

3.6.1. UDP

可以用于 RPC 等.

3.6.1.1. 段格式

精品背书课.

3.6.2. TCP

3.6.2.1. 可靠数据传输机制

1. 序号机制: 每个字节都有编号, 数据段的编号是第一个字节的编号.
2. 确认机制: 确认号是期望收到的下一个数据段的序号, 也即已经成功接收的数据号的后继. TCP 使用累计确认, 如果确认了后来的数据, 表示前面的数据也已经成功接收. 数据中可以携带确认号, 这叫做捎带确认(piggybacking).
3. 重传机制
   1. 计时器超时: 太长时间没收到确认就重传, 超时时间设置为预估的往返时间
   2. 三次重复确认: 如果中间一个丢失了, 后面的正常发送, 于是对于后面正常接收的每一个, 都会回复一个丢失包的序号的确认, 三次重复确认就会使发送方重传. 注意是三次**重复**, 所以事实上应该有四次都是一样的 ACK.

3.6.2.2. 滑动窗口流量控制

TCP 是全双工传输, 所以两方都各自维护自己的发送窗口和接收端口.

所谓窗口也即发出去了但是对方还没回复(自己还没处理完)的包的最大长度, 双方可以用数据段里面的接收窗口值告知对方调整发送窗口.

糊涂窗口综合征: TCP 流控不良, 窗口非常小, 头部显得非常大, 额外开销过高. 解决方案有

1. Nagle 算法: **发送方**先收集要发送的小数据, 到达一定量或者收到对方确认之后再发送.
2. Clark 算法: **接收方**收到数据段就确认, 但是直到缓冲区足够大之前一直宣布窗口为 0, 也即暂停接收.

拥塞窗口: 发送方维护, 用于限制自己, 不需要通告接收方. 实际的发送窗口是接收窗口和拥塞窗口的最小值.

例 3.17

如果一个 TCP 数据段的接收方, 向发送方发出了一个确认消息, 其中的 ACK 确认号和窗口尺寸两个字段的值分别是: ACK=12000, WIN=8000. 下列哪一项不是发送方的可以传输的有效的数据段?

1. 发送方可以传输 2000 字节数据段, SEQ = 18100
2. 发送方可以传输 1500 字节数据段, SEQ = 18100
3. 发送方可以传输 1000 字节数据段, SEQ = 18000
4. 发送方可以传输 2000 字节数据段, SEQ = 17000

解

接收方确认到 11999, 期望 12000 的时候还有 8000 的窗口, 所以说新发给他的数据不能超过 8000, 也即末尾序号不能超过 $12000 + 8000 - 1 = 19999$, 显然第一个选项超过了.

3.6.2.3. 连接的建立

也即所谓三次握手:

1. 客户端向服务端, SYN 设置为 1, seq 值设置为随机值 x, 数据段无数据, 但是占一字节
2. 服务端向客户端, SYN 和 ACK 设置为 1, seq 为随机值 y, ack 为 $x + 1$, 数据段无数据, 但是占一字节
3. 客户端向服务端, ACK 设置为 1, seq 为 $x + 1$, ack 为 $y + 1$, 这次可以包含数据, 也可以不包含, 也即数据长度为 0

3.6.2.4. 连接的释放

分为

- 对称释放: 两方独立完成连接释放并收到对方确认
- 非对称释放: 只要一方完成连接释放的,请求并收到确认即可

3.6.2.4.1. 非对称释放

断开一方直接发送 DR 并关闭连接.

3.6.2.4.2. 对称释放

所谓四次挥手.

1. 客户端向服务端, FIN 设置为 1, seq 为 x(并非随机, 而是真实值), 数据占用 1 字节
2. 服务端向客户端, ACK 为 1, seq 为 y, ack 为 $x + 1$, 此时客户端向服务端的连接已经关闭.
3. 服务端向客户端, FIN 为 1, ACK 为 1, seq 为 z(不一定是 $y + 1$, 可能在单向连接关闭之后又发了一些), ack 为 $x + 1$, 数据占用 1 字节
4. 客户端向服务端, ACK 为 1, seq 为 $x + 1$, ack 为 $z + 1$, 然后等待两倍的最长段寿命(MSL, 一般为 2min)后服务端向客户端的连接正式关闭

3.6.2.5. 计时器

TCP 的状态需要多种计时器维护, 包括

3.6.2.5.1. 重传计时器

太长时间没收到确认就重传, 超时时间设置为预估的往返时间(RTT).

RTT 的估计使用指数加权移动平均, 也即逐渐修正

$$\mathrm{SRTT} \leftarrow (1 - \alpha)\ \mathrm{SRTT} + \alpha R$$

其中 α 是平滑因子, 越大则更新越平滑, 一般取 $\alpha = \frac{1}{8}$.

也有其他的估计方法.

3.6.2.5.2. 持续计时器

接收方有时候会把窗口设置为 0 主动停止接收信息, 直到他再次把窗口设置为其他值, 但是有时候还原窗口的消息丢失了, 这就会造成暂停状态无法解除.

所以我们设置了持续计时器, 如果**发送方**收到的消息中 rwnd 设置为 0, 则开启计时器, 超时时则主动发送一个探测数据段(只有 1 字节新数据)触发对方重发一个确认数据段(which 重新设置了 rwnd), 并且同时再启动一个新的持续计时器.

虽然探测数据段中有 1 字节新数据, 但是我们必须考虑接收方不会接收它(毕竟窗口 officially 还是 0), 所以我们后续的数据段的 seq 忽略这 1 字节.

3.6.2.5.3. 保活计时器

防止长期空连接.

每收到一次客户端数据段就重置计时器, 超时的时候**服务端**就主动发送一个探测数据段, 连续 10 个探测数据段都没收到回应则认为客户端失联, 主动关闭连接.

3.6.2.5.4. 时间等待计时器

主要用于连接释放阶段, 也即前面提到的等待两倍的最长段寿命(MSL, 一般为 2min)后服务端向客户端的连接正式关闭.

3.6.2.6. 拥塞避免

主要采用加法增, 乘法减的思想.

- 慢启动阶段: 有一个初始拥塞窗口大小, 比如 1 倍 MSS(最大段数据), 每次发完收到对方确认后, 就把拥塞窗口加倍
- 拥塞避免阶段: 慢启动达到阈值(窗口达到阈值)后, 进入线性增加阶段, 每次增加 1MSS
- 网络拥塞事件发生阶段: 发生拥塞, 把阈值设置为当前拥塞窗口的一半, 然后拥塞窗口设置为 1, 然后重新进入慢启动阶段
- **快速重传**和恢复阶段: 如果只有个别数据丢失(比如三次重复确认), 就立即重传, 把阈值设置为窗口的一半, 窗口设置为新的阈值, 然后进入拥塞避免阶段

所以此处快速重传也属于拥塞控制手段.

3.6.2.7. TCP 状态

有如下状态

状态	含义
CLOSED	没有活跃的连接或者挂起
LISTEN	服务器等待入境呼叫
SYN RCVD	到达一个连接请求, 等待 ACK
SYN SENT	应用已经启动了打开一个连接
ESTABLISHED	正常的数据传送状态, 既可以接收也可以发送
FIN WAIT1	应用没有数据要发了
FIN WAIT2	另一端同意释放连接
TIME WAIT	等待所有数据包寿终正寝
CLOSING	两端同时试图关闭连接

CLOSE WAIT	另一端已经发送关闭连接
LAST ACK	等待所有数据包寿终正寝

3.6.3. QUIC

新型协议, 基于 UDP.

优势包括:
- 基于 UDP, 延迟低
- 传输性能高
- 安全, 默认 TLS 1.3
- 解决 TCP 队头堵塞问题
- 提供可插拔的拥塞控制机制
- 可以连接迁移

3.7. 应用层

3.7.1. 域名系统

arpa 不是常见的顶级域名(TLD).

根服务器管理整个域名树.

域名服务器分为
- 主服务器: 创建维护更新 zone 数据
- 备份服务器: 只从主服务器获取数据

DNS 解析有两种方式
- 递归解析: 一级一级深入, 每级都返回下一级服务器的结果
- 迭代解析: 不断询问另一个服务器下一个要询问的服务器

3.7.2. 典型应用

3.7.2.1. 文件传输

FTP 协议, 20 是数据端口, 21 是控制端口.

3.7.2.2. 远程登录

- telnet: 简单的远程终端协议, 最大问题是不安全
- SSH: 安全

3.7.2.3. 电子邮件

包括用户代理(UA), 邮件传输代理(MTA), 和邮件发送协议.

发送用 SMTP, 接收用 POP3 或者 IMAP.

IMAP 和 POP3 不同, POP3 就是一个邮件转发器, 而 IMAP 就像一个网盘一样, 用户可以像在本地操作一样操作服务器上的 IMAP 邮箱内容.

发邮件用不到对方的 UA, 只用自己的 UA, 所以只涉及一个 UA.

3.7.3. 万维网

万维网由资源, 统一资源定位符, 传输协议组成; 基本组成元素是 Web 服务器, Web 浏览器, HTML.

URL 是统一资源定位符, 给各种资源一种唯一的, 抽象的识别方法.

在客户/服务器模型中, 客户和服务器程序相互独立.

前端压力过大可以用 TCP 移交直接让后端服务器处理, 缓解前端压力.

3.8. 其它

3.8.1. 各协议 PDU 及相关数值

- 物理层: 比特流
- 链路层: 帧
  - 以太网

    每行 22 字节

    <table>
    <tr><td colspan="3">前导码8B</td><td colspan="3">目的地址6B</td><td colspan="3">源地址6B</td><td>长度2B</td></tr>
    <tr><td colspan="10">数据+填充 46-1500B</td></tr>
    <tr><td colspan="2">校验和4B</td></tr>
    </table>

    这是 IEEE 802.3 格式的帧, 现在通行的 DIX 在前导码上有略微不同, 且长度字段为类型.
  - WiFi(IEEE 802.11)

    每行 32 字节

    <table>
    <tr><td>帧控制 2B</td><td>持续时间 2B</td><td colspan="2">地址1 6B</td><td colspan="2">地址2 6B</td><td colspan="2">地址3 6B</td><td>序号控制 2B</td><td colspan="2">地址4 6B</td><td>长度 2B</td></tr>
    <tr><td colspan="12">数据 0-2312B</td></tr>
    <tr><td colspan="2">校验序列 4B</td></tr>
    </table>

    其中帧控制部分有 11 个子字段

    每行 16 位

    <table>
    <tr><td colspan="2">协议版本</td><td colspan="2">类型</td><td colspan="3">子类型</td><td>去往 DS</td><td>来自 DS</td><td>更多分片</td><td>重试</td><td>电源管理</td><td>更多数据</td><td>加密保护</td><td>顺序</td></tr>
    </table>

- 网络层: 分组
  - IPv4

    每行 32 位

    <table>
    <tr><td colspan="2">版本 4b</td><td colspan="2">头部长 4b</td><td colspan="3">区分服务类型 8b</td><td colspan="8">总长度 16b</td></tr>
    <tr><td colspan="7">标识(分组序号) 16b</td><td></td><td>DF</td><td>MF</td><td colspan="5">片偏移 13b</td></tr>
    <tr><td colspan="3">TTL 8b</td><td colspan="4">协议 8b</td><td colspan="8">头部校验和 16b</td></tr>
    <tr><td colspan="15">源IP 32b</td></tr>
    <tr><td colspan="15">目的IP 32b</td></tr>
    <tr><td colspan="15">选项...</td></tr>
    </table>

载荷

- 头部长度: 单位为 4 字节
- 总长度: 单位为字节
- 片偏移: 单位为 8 字节

▸ IPv6

每行 32 位

版本	流量类型	流标签	
载荷长度		下一个头	跳数限制
源 IP 地址			
源 IP 地址			
源 IP 地址			
源 IP 地址			
目的 IP 地址			
目的 IP 地址			
目的 IP 地址			
目的 IP 地址			
0-6个扩展头			
载荷			

- 下一个头代替了协议
- 扩展头有
  - 逐跳选项
  - 路由头 RH
  - 分片头 FH
  - 认证拓展头 AH
  - 目的选项拓展头
  - 封装安全载荷拓展头 ESP(但是属于 IPSec 协议族, 不属于 IPv6)

• 传输层

▸ TCP: 段

每行 4 字节

源端口号								目的端口号
序号seq								
确认号ack								
头部长度	保留	URG	ACK	PSH	RST	SYN	FIN	接收窗口

段校验和	紧急指针
选项 $n \times 4$ Bytes	
数据	

- UDP: 数据报

  每行 8 字节

源端口号	目的端口号	总长度	数据报校验和

相关数值

- 头部开销
  - IP: 20 Bytes
  - TCP: 20 Bytes
- 最小传输单元
  - 以太网(不含前导码): 64 Bytes (出于 CSMA/CD 要求)
  - IP(MTU): 576 Bytes
- 最大帧长
  - 以太网(不含前导码): 1518 Bytes
- 最大头部长度
  - IPv4: 64 Bytes, 因为头部长度字段单位为 4 Bytes, 本身占 4 bits
- 最大有效载荷
  - IP: 65515 Bytes
  - TCP: 65495 Bytes
    - 这是因为分组总长度最大 65535 Bytes (分组的长度字段为 16 位), 减去 IP 分组和 TCP 的两个 20 字节的头部, 也即 65495 Bytes

3.8.2. 广播域和冲突域

路由器的一个端口就是一个广播域和一个冲突域.

交换机能分割冲突域, 集线器不能.

3.8.3. 思科路由器

路由器 NVRAM 里面存的是配置.

- 进入特权模式

```
en
```

- 进入全局配置模式(在特权模式下)

```
config t
```

- 显示路由表

```
show ip route
```

3.8.4. 接口配置

- 进入接口配置模式(在全局配置模式)

```
interface fax/x
```

- 配置 IPv4 地址

```
ip address IP 子网掩码
```

- 配置 IPv6 地址

```
ipv6 address IP 子网掩码
ipv6 enable
```

- 启动接口

```
no shutdown
```

3.8.5. 路由配置

- 配置静态路由

```
ip route 目标IP 子网掩码 下一跳IP
```

删除只需要在这条命令前面加 `no`.

- 配置 RIP(在全局配置模式下)

```
router rip
network IP
```

指定 IP 对应网络可以接受 RIP 消息, 关闭 RIP 只需要在第一条命令前面加 `no`.

- 配置 OSPF(在全局配置模式下)

```
router ospf 进程号
network IP 通配掩码 area 区域号
```

进程号可以取 10, 100 等, 区域号写 0 即可, 表示骨干区域. 通配掩码是子网掩码的反演.

指定 IP 对应网络可以接受 OSPF 消息, 关闭 OSPF 只需要在第一条命令前面加 `no`.

OSPF 和 RIP 都只需要把本路由器各端口对应的网络使用 `network` 配置.

2025-05-25

团精确计数的 Pivoter 算法

对 Pivoter 算法的介绍, 包括关键数据结构 SCT 的构建和性质, 以及如何使用 SCT 来计数团.

这是我“数据结构”课程大作业的一部分内容, 由于中文网络上似乎没有 Pivoter 算法的相关资料, 所以我整理成一篇文章来介绍这个算法.

4.1 任务

我所面临的任务是: 在给定的无向图中, 计算所有团(clique, 也即完全子图)的个数. 除此之外, 常见的任务还可以是计算某些点或者某些边参与的团的个数, 或者把这些团列出来.

方便起见, 下面默认任务是计算所有团的个数.

4.2 记号

为了防止混淆, 本文中递归树结构的边称为“连接”, 而图中的边称为“边”, 递归树结构的点称为“节点”, 而图中的点称为“顶点”.

记号	含义
$G(V, E)$	无向图, 其中 V 是顶点集, E 是边集
n	顶点数, 即 $\|V\|$
m	边数, 即 $\|E\|$
α	退化度, 也即图中最大团的大小
C	团, 也即完全子图
C_k	k 阶团, 也即包含 k 个顶点的团
$N(v)$	顶点 v 的邻居集
$N^+(v)$	顶点 v 的出邻居集, 对有向图或无向图的一个定向适用
$N(S, v)$	顶点集 S 中顶点 v 的邻居集, 即 $N(v) \cap S$
$\mathrm{SCT}(G)$	图 G 的简化团树, 即 Pivoter 算法的递归树
$\boldsymbol{T}$	SCT, 也即简化团树
$\mathfrak{h}$	SCT 中连接标签的 hold 类型
$\mathfrak{p}$	SCT 中连接标签的 pivot 类型
T	递归树中的一条路径, 也即从根到某个叶子节点的路径
$H(T)$	T 中标签类型为 $\mathfrak{h}$ 的所有边的标签中的节点的集合
$P(T)$	T 中标签类型为 $\mathfrak{p}$ 的所有边的标签中的节点的集合

表 1　符号表

4.3 Pivoter 算法

4.3.1 Pivoter 算法简介

Pivoter 算法是 Shweta Jain 等人于 2020 年提出的一种精确计数团的算法, 原论文发表在 WSDM 2020 上.

Pivoter 算法借鉴了 BK 算法的 Pivot 思想, 通过选择一个顶点作为 Pivot 构建一种具有良好性质的新的递归树结构 SCT(Succinct Clique Tree, 简化团树) 大大减少了朴素递归算法的时空代价. Pivoter 算法的时间复杂度是 $O(\alpha^2|\mathrm{SCT}(G)| + m + n)$, 空间复杂度是 $O(m + n)$.

4.3.2 Pivoter 算法的思想

4.3.2.1 朴素递归算法

我们可以先从朴素递归算法开始, 它的思路是: 所有包含 v 的团都可以通过把 v 添加到另一个团中得到, 后者所提到的团必定在 $N(v)$ 中.

所以我们可以通过下面的这个递归过程找到所有团:

```
GetCliques(V)
    If |V| ≤ 1:
        Return [[v]]
    R ← []
    For v in V:
        ℂ ← GetCliques(N(v))
        ℂ ← ℂ ∪ {C ∪ {v} | C ∈ ℂ}
        R ← R ∪ ℂ
    Return R
```

我们可以想到朴素递归算法的递归树:

- 每个节点都对应一个 V 的子集 S, 大多数时候是某个顶点的邻居, 我们称之为节点的标签
- 标签为 S 的节点的出连接对应了所有 $s \in S$, 我们称之为连接的标签

于是我们可以看出, 从根路径出发向下的任何路径(可以不到叶子节点)都给出一个团, 也即这条路径途径的所有连接的标签, 而且我们能通过这种方式获得所有团. 这是因为

1. 一个标签为 v 的节点的子树中的所有节点(包括子节点, 子节点的子节点, ...)显然都在 $N(v)$ 中, 也即他们都和 v 有边. 对路径上的所有节点都应用这个观察即可得知这是一个团
2. 任取一个团 $v_1, v_2, v_3 ... v_n$, 显然 v_1 标记的连接必定在递归树的根节点之下, 而 $v_2 ... v_n$ 都是 v_1 的邻居, 他们必定在这一连接所连接的另一个节点的标签中, 然后便可以反复应用这一规则, 直到说明所有 n 个节点都在树上, 而且呈一条向下的路径排列

但是显然这种方式会导致重复生成团, 一种避免的方式是对这个图生成一个有向无环图(DAG), 然后把 N 修改为只给出出邻居节点.

但是即使如此, 在大型图上我们也不能完整地构建这个递归树. 所以 Pivoter 算法试图找出一种压缩这棵树, 而且获得所有团的唯一表示的方式.

4.3.2.2 Pivoter 和 SCT

对于上述递归树中的一个标签为 S 的节点, 我们取出一个轴节点 $p \in S$, 于是 S 中的任一团 C 可以被分为三类

1. $p \in C$

2. $C \subset N(p)$
3. C 中有一个 p 的非邻居

前两类之间有一一对应的关系: 把每个第一类团里面的 p 拿掉就可以得到第二类团, 反之亦然, 于是我们这里可以只在树中继续处理第二类团.

对于这个节点, 我们递归调用以获取在 $N(p) \cap S$ 和 $N(u) \cap S$ 中的团, 其中 u 是 S 中 p 的非邻居节点. 这两种团分别对应前面所说的第二类和第三类.

这里 p 可以取度最高的节点, 这是因为节点的度越大, 他就更有可能在更多团中, 那我们省下的第一类节点的代价就越多.

按这种算法构造的递归树就是所谓 SCT. SCT 可以轻易在千万级的图上构建.

4.3.2.3 使用 SCT 进行团计数

和一般递归算法的递归树相比, SCT 的每条路径 T 的连接标签集仍然对应一个团, 但是不是所有团都有路径对应, 实际上, 这也是为什么一般递归算法的递归树如此巨大.

但是如果这样, 我们又如何使用 SCT 来计数所有团呢? 实际上这正是 Pivoter 算法的关键贡献所在: SCT 有一种良好的性质, 使得我们可以在图中找到所有团的唯一表示. 而我们正是通过这种方式来计数所有团的. 至于具体如何, 我们将在接下来的内容中介绍.

4.3.3 Pivoter 算法的实现

4.3.3.1 SCT 的构建

在此之前, 我们要先具体定义 SCT 的结构:

- SCT 是一棵树.
- SCT 的每个节点都有一个标签, 这个标签是一个顶点集. 其中根节点的标签是 V.
- SCT 的每个连接都有一个标签, 这个标签是一个顶点-类型对, 也即 $(v, \cdot)$, 其中 v 是该连接接近根节点一端的节点标签中的一个元素, 类型为 $\mathfrak{p}$ 或者 $\mathfrak{h}$.

下面我们给出一个 SCT 的构建算法, 这是一个 BFS 算法.

```
BuildSCT(G):
    N⁺ ← BuildDAG(G)
    Init T with root node R labeled by V
    Q ← empty queue
    For v in V:
        N ← new node labeled by N⁺(v)
        T ← T with link labeled by (v, 𝔥) from R to v
        Push N into Q
    While Q is not empty:
        P ← pop node from Q
        S ← label of P
        If S is ∅:
            Continue
        p ← arg max_p |N(S, p)|
        N ← new node labeled by N(S, p)
        T ← T with link labeled by (p, 𝔭) from P to N
        Push N into Q
```

18 | | $\{v_1, v_2, ..., v_l\} \leftarrow S \setminus (p \cup N(p))$
19 | | **For** $i < l$:
20 | | | $\mathcal{N}_2 \leftarrow$ new node labeled by $N(S, v_i) \setminus \{v_1, v_2, ..., v_{i-1}\}$
21 | | | $\boldsymbol{T} \leftarrow \boldsymbol{T}$ with link labeled by $(v_i, \mathfrak{h})$ from $\mathcal{N}$ to $\mathcal{N}_2$
22 | | | Push $\mathcal{N}_2$ into Q
23 | **Return** $\boldsymbol{T}$

这其中新建到 $\mathcal{N}$ 的连接和新建到 $\mathcal{N}_2$ 的连接分别对应了上面所说的第二类和第三类团.

4.3.3.2 SCT 的性质及证明

SCT 的唯一编码性 对于图和在图上构建的 SCT $\boldsymbol{T}$, 图中的每个团都能被唯一表示为 $H(T) \cup Q$. 其中 $Q \subseteq P(T)$, T 是 SCT 上一条从根到叶的路径.

我们可以看出, 每个从根到叶的路径都表示一个团, 也即 $H(T) \cup P(T)$. 其他的团可能就是这样的集合的子集. 这样的子集可能会多次出现, 但是如果我们考虑 $H(T)$ 和 $P(T)$ 的不同, 也即考虑连接的类型, 那么这种表示就是唯一的, 这也正是这个定理想要描述的内容.

证明:

考虑标签是 S 的节点 γ. 接下来我们将通过归纳 $|S|$ 证明, 每个团 $C \subseteq S$ 都可以表示为 $H(T) \cup Q$. 其中 T 是一条从 γ 到叶子的路径, $Q \subseteq P(T)$.

这里对 SCT 的归纳是从下向上的(对应 $|S|$ 从小变大), 每次添加一个父亲节点到顶上, 下面归纳情况中我们添加的其实就是前面提到的节点 γ.

1. 基本情况: S 是空的, 其他所有相关内容都是空的, 自然成立.
2. 归纳情况: 令 p 为构建 SCT 时运行到 γ 节点时选择的轴. 对于所有的团, 我们有三种情况, 分别对应了 SCT 构建时选取轴之后的三种情况.
   1. $p \in C$. 也即 γ 之下有一个标签为 $(p, \mathfrak{p})$ 的连接, 连接到 γ 的一个子节点 β. β 有标签 $N(S, p)$.
      - 存在性: 观察到 $C \setminus p$ 是 $N(S, p)$ 中的一个团. 由归纳假设, $C \setminus p$ 有一个唯一表示 $H(T) \cup Q$, 其中 T 是从 β 到叶子的路径且 $Q \subseteq P(T)$. 而且 C 没有一种这样的表示(利用从 β 到叶子的路径的), 因为 $N(S, p) \not\ni p$.

        令 T' 为包含路径 T 而且从 γ 开始的路径, 则有 $H(T') = H(T)$, $P(T') = P(T) \cup p$. 于是可以把 C 表达为 $H(T') \cup (Q \cup p)$, 其中 $Q \cup p \subseteq P(T')$.
      - 唯一性: 也即我们需要说明没有其他路径可以表示 C. 我们可以讨论上述表示中使用的路径 T
        1. T 没有经过 β: 考虑任意从 γ 开始, 但是不经过 β 的路径, 它一定经由标签为 $(v_i, \mathfrak{h})$ 的路径经过了其他子节点, 其中 v_i 是 p 的非邻居. 那么由于 $p \in C$, p 的非邻居显然不在 C 中, 所以利用这样的路径无法表示 C.
        2. T 经过 β: 如果是经过了 β 的路径, 根据归纳假设即可知其唯一.
   2. $C \subseteq N(S, p)$. 这种其实就是第一种情况的 C 去除 p, 区别就是新增节点不在 C 中. 所以表示也和第一种情况类似, 也即复用之前的表示的节点选取(从而没有加入 p), 但是使用包含 γ 的路径.
   3. C 包含一个 p 的非邻居节点.

我们先重复代码的步骤把各种符号都恢复出来: 还是令 $\{v_1, v_2, ..., v_l\} = S \setminus (N(p) \cup p)$, 令 $i = \arg\min_{j;v_j \in C} j$, 也即 v_i 是这些 v 中首个包含在 C 中的节点. 对于任意的 $1 \leq j \leq l$, 令 $N_j := N(S, v_j) \setminus \{v_1, v_2, ..., v_{j-1}\}$. 根据我们生成 SCT 的步骤, γ 下面有 l 个标签为 N_j 的孩子节点, 而且这些节点到 γ 的连接都是 $\mathfrak{h}$ 类型的, 所以对于经过 N_j 的路径 T, 有 $H(T) \ni v_j$. ($\mathfrak{h}$ 型连接的标签的节点必定选取)

- 存在性: 我们可以讨论将要在表示中使用的路径 T
  1. T 经过 $N_j, j < i$: 如果能利用 T 表示 C, 它就不能经过 $j < i$ 的 N_j, 因为 v_i 才是首个包含在 C 中的节点, v_i 之前的均不在 C 中却被选取.
  2. T 经过 $N_j, j > i$: 如果 $j > i$, 那 $N_j \not\ni v_i$ 更是没有路径能表示 C, 这是因为前面包含在 C 中的 v_i 被去掉了.
  3. T 经过 $N_j, j = i$: 由上述两种情况, 我们知道如果能利用一条路径表示 C, 那它一定经过 N_i. 注意到 $C \setminus v_i$ 是在 N_i 中的团, 由归纳假设, 有 $C \setminus v_i$ 的唯一表示 $H(T) \cup Q$, 其中 $Q \subseteq P(T)$, T 是从 N_i 开始到叶子的路径. 令 T' 为包含路径 T 而且从 γ 开始的路径, 有 $H(T') = H(T) \cup v_i$, 所以 $C = H(T') \cup Q$.
- 唯一性: 显然.

证毕.

另一方面, 每一个合法的表示显然都对应了一个团. 这其实与简单递归算法的递归树的情形类似: 因为父节点到子节点的连接标签的节点是父节点标签中的一个节点, 而子节点的标签都是他的邻居, 而且还是还是父节点标签的子集, 所以完整的路径都能表示一个团, 从里面去除一部分 p 也是如此.

4.3.3.3 利用 SCT 的唯一编码性进行团计数

根据上述关键定理, 一条从根到叶的路径 T 表示了 $2^{P(T)}$ 个不同的团. 其中大小为 $|H(T)| + i$ 的有 $\binom{P(T)}{i}$ 个, 所以我们可以轻易得出如下算法.

```
Pivoter(G):
    T ← BuildSCT(G)
    R ← [0, 0, ...]
    For T in T:
        For i ≤ |P(T)|:
            R[|H(T)| + i] ← R[|H(T)| + i] + C(P(T), i)
    Return R
```

当然, 这只是针对我的任务, 也即全局团精确计数的 Pivoter 算法. 对于边或者顶点参与的团计数, 或者列出所有团, 只需要在上述算法中稍作修改即可.

4.3.4 Pivoter 算法的复杂度分析

参见原论文.

4.3.5 Pivoter 算法的代码实现

我在网络上找到了两个 Pivoter 算法的实现:

- 论文作者给出的 C 语言实现: Bitbucket 仓库
- charunupara 给出的 Julia 实现: GitHub 仓库

另外, 我基于论文作者的 C 语言实现, 修改了一个只有全局团计数的 C++ 实现: GitHub 仓库.

不过这些实现都没有提供并行化的版本. 一些更新的算法的论文中提到了他们的算法和并行 Pivoter 算法的比较, 但是并没有提供代码.

2025-05-19

判定我变老的标准

当我不加思考地贬低和否定新事物, 不再认为情况会有所改善时, 我就变老了.

2025-04-14

近乎完美的 GitLab + frp 搭建踩坑

使用 GitLab 和 frp 搭建私有服务器的经验分享

很早之前就想自建一个 Git server, 终于在这个月早些动工了.

我的基本要求是
- 一个公网可以访问的, 全链路 HTTPS 保护的 GitLab 网站
- 可以正常使用 Git over SSH
- 有邮箱通知(尽管可能被标记为垃圾邮件)
- 服务器的 IP 不被泄露, 但是不支付额外费用购买如 Cloudflare 等平台的付费服务

为此, 我使用了
- 一台我有 root 权限的公网服务器, 用于运行 frp, wstunnel 等
- 一台我有 root 权限的内网服务器, 用于运行 GitLab 本体
- 一个二级域名
- 一个支持 SMTP 的邮箱(Gmail)

6.1 思路

基本结构如下:

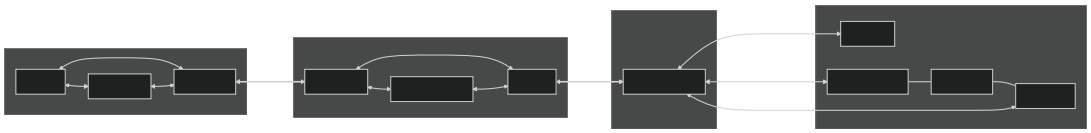

图 20　基本结构

个人认为比较重要的部分就是通过 wstunnel 代理 Git over SSH 的流量, 于是可以走 Cloudflare 达到不泄露服务器 IP 的目的.

6.2 部署过程

在开始之前, 我们先假设一些值用作示例:
- 我们给 GitLab 服务预留的域名是 gitlab.example.com
- wstunnel 服务器的 path prefix 是 ssh-tunnel
- 公网服务器的 IP 是 1.2.3.4
- 公网服务器暴露了以下端口

端口	服务
10001	frp server
80	nginx http
443	nginx https

- 公网服务器上还预留了以下端口, 但是不需要暴露

端口	服务
10002	GitLab Web
10003	GitLab SSH
10004	wstunnel server

- 内网服务器上预留了以下端口

端口	服务
801	GitLab Web
221	GitLab SSH

6.2.1 FRP Server 的安装和配置

在公网服务器上进行.

从 frp 的官方仓库下载即可.

这里提供一个简单的配置.

```
bindPort = 10001
auth.token = "some-random-password"
```

启动 frps

```
frps -c config.toml
```

6.2.2 wstunnel Server 的安装和配置

在公网服务器上进行.

从 wstunnel 的官方仓库下载即可.

可以一行启动:

```
wstunnel server ws://0.0.0.0:10004 --restrict-http-upgrade-path-prefix ssh-tunnel --
restrict-to localhost:10003
```

这里按前面所说的额外限制了 path perfix 和可以访问的端口以提高安全性.

6.2.3 GitLab 安装和配置

在内网服务器上进行.

遵循官方教程即可.

需要注意我们使用 SMTP 提供邮件服务, 所以不需要安装 postfix. 此外, 安装时首次 configurate 申请证书会失败, 这无关紧要, 我们之后使用自签名证书即可.

编辑 `/etc/gitlab/gitlab.rb` 中的这些配置项

- `external_url`: 如果你没有在安装时指定, 现在可以设置了, 我们提到过使用 `https://gitlab.example.com` 作为示例
- SMTP 相关配置: 我这里根据官方文档中的指引进行配置. 我还额外设置了 `gitlab_rails['gitlab_email_from']` 为真实的邮箱用户名.
- `letsencrypt['enable']`: 修改为 `false`. 我已经解释过.

- `nginx['listen_port']`: 如果你的服务器上还运行了其他占用 80 端口的服务, 可以把这个端口修改为其他端口, 这里使用 801 作为示例.

生成自签名证书

```
cd /etc/gitlab/ssl
sudo openssl req -x509 -nodes -days 3650 -newkey rsa:2048 -keyout
gitlab.example.com.key -out gitlab.example.com.crt
sudo chmod 600 gitlab.example.com.*
sudo chown root:root gitlab.example.com.*
```

做完这些后, 可以重新配置和重启 GitLab:

```
sudo gitlab-ctl reconfigure
sudo gitlab-ctl restart
```

6.2.4 sshd 的安装和配置

在内网服务器上进行.

直接从系统软件源安装即可.

安全起见, 我们使用一个单独的端口暴露 Git over SSH 服务, 并且只允许 git 用户登录, 这里使用 221 端口作为示例.

在 `/etc/ssh/sshd_config` 添加

```
Port 221
Match LocalPort=221
  AllowUsers git
```

重启 SSHD 服务

```
sudo systemctl restart sshd
```

6.2.5 FRP Client 的安装和配置

在内网服务器上进行.

从 frp 的官方仓库下载即可.

然后对我们用到的 nginx 和 SSH 端口做转发:

```
serverAddr = "1.2.3.4"
serverPort = 10001
auth.token = "some-random-password"

[[proxies]]
name = "gitlab-web"
type = "tcp"
localPort = 801
remotePort = 10002

[[proxies]]
name = "gitlab-ssh"
type = "tcp"
```

```
localPort = 221
remotePort = 10003
```

启动 frpc

```
frpc -c gitlab.toml
```

6.2.6 Cloudflare 的配置

在 Cloudflare 中解析 gitlab.example.com 到你的公网服务器即可.

6.2.7 nginx 的安装和配置

在公网服务器上进行.

直接从系统的软件源安装即可.

我们需要获取内网服务器上 GitLab 服务使用的自签名证书

```
sudo bash -c 'echo | openssl s_client -connect localhost:10002 -servername
gitlab.example.com 2>/dev/null | openssl x509 > /etc/nginx/ssl/gitlab_cert.pem'
```

此外, 我们还需要使用 acme.sh 申请证书, 遵从其官方指引即可, 这里不再赘述.

这里给出一个 `/etc/nginx/conf.d/gitlab.conf` 的参考配置, 具体可以根据实际情况再修改:

```
server {
    listen 443 ssl;
    server_name gitlab.example.com;

    ssl_certificate /root/.acme.sh/gitlab.example.com_ecc/gitlab.example.com.cer;
    ssl_certificate_key /root/.acme.sh/gitlab.example.com_ecc/
gitlab.example.com.key;

    location / {
        proxy_pass https://localhost:10002;

        proxy_ssl_verify off;
        proxy_ssl_trusted_certificate /etc/nginx/ssl/gitlab_cert.pem;
        proxy_ssl_verify_depth 2;

        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_read_timeout 300;
        proxy_connect_timeout 300;
        proxy_send_timeout 300;
    }

    location /ssh-tunnel {
        if ($http_upgrade = "") {
            return 404;
```

```
        }
        proxy_redirect off;
        keepalive_timeout 12000s;
        proxy_pass http://127.0.0.1:10004;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Host $host;
        proxy_set_header Connection "upgrade";
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_intercept_errors on;
        proxy_pass_request_headers on;
    }
}

server {
    listen 80;
    server_name gitlab.example.com;

    location / {
        proxy_pass http://localhost:10002;
        proxy_ssl_verify off;
        proxy_http_version 1.1;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_read_timeout 300;
        proxy_connect_timeout 300;
        proxy_send_timeout 300;
    }
}
```

6.2.8 客户端需要的额外配置

需要安装 wstunnel, 并且在 `~/.ssh/config` 中额外添加

```
Host gitlab.example.com
    ProxyCommand=wstunnel client wss://gitlab.example.com/ssh-tunnel --http-upgrade-
path-prefix ssh-tunnel -L stdio://127.0.0.1:10003
```

6.3 Troubleshooting

一些问题的诊断方法, 和我遇到的一些问题的解决方案.

6.3.1 我不知道 root 用户的密码

如果你忘记了在安装时指定 root 用户密码, 可以遵照 官方文档 中的方法修改密码.

6.3.2 无法访问服务

由内而外地诊断哪里出了问题. 比如先在内网服务器上 `curl http://localhost:801`, 测试通过再在公网服务器上测试 `localhost:10002`, `localhost:80`.

Git over SSH 的问题也可以以同样的方式诊断.

6.3.3 GitLab 无法保存配置, 错误代码 500

看上去是一些 token 错误引起的问题, 总体的解决方案就是删掉这些 token. 我尝试了若干方法, 已经不太清楚哪一步起了作用, 这里是我当时查阅过的内容:

- https://gitlab.com/gitlab-org/gitlab/-/issues/419923
- https://gitlab.com/gitlab-org/gitlab/-/issues/334862
- https://gitlab.com/gitlab-org/gitlab/-/issues/301170
- https://forum.gitlab.com/t/500-error-access-admin-runners-not-a-migration/100875

6.3.4 注册邮件无法正常发送

可以按照 GitLab 官方文档 中的步骤进行诊断.

如果最后发现 `Notify.test_email` 无法正常发信, 可以使用第三方工具比如 swaks 按相同的配置发信看看有无更详细的错误信息.

2025-02-28

宝宝的强化学习

简单的强化学习入门, 包括马尔可夫决策过程, Q-Learning 和 DQN 算法的介绍和实现.

你可能或多或少听说过(或者十分了解)“监督学习”, 也就是给出很多自变量-因变量对, 然后通过某种方式拟合那个函数.

但是监督学习并不能解决所有问题, 尤其是当我们没有那么多自变量-因变量对时候. 如果能把我们的问题转换成模型做出一些“动作”, 从而和“环境”交互, 而且我们能较轻易地得知交互结果的好坏(无论是最终地还是暂时地), 我们能不能从这些结果的好坏中获得经验, 从而优化模型的行为呢?

7.1 环境? 动作? 结果?

我们暂时承认这个想法有可能是可行的, 但是为了搞清楚他是不是真的可行, 我们需要做一些严肃的推理和实验.

因此, 我们要先**形式化**这个问题.

我们用一个状态量 $s \in S$ 来描述环境, 模型采取的行动 $a \in A$ 会让 s **改变**, 而且每次改变, 模型都会得到 $r(s, a)$ 的奖励(或者惩罚). 具体是怎么**改变**呢, 比较好(通用)的描述应该是产生一个下一状态的概率分布 $P(s' \mid s, a)$.

由于这显然是一个时序的过程, 所以我们定义时间步 $t < T$, 另外有在第 t 步时的状态为 S_t, 模型行动为 A_t, 获得的奖励为 $R_t = r(S_t, A_t)$.

为了控制模型对近期和远期利益的倾向, 我们引入一个折扣因子 γ, 并且定义从时刻 t 开始直到结束, 模型获得的总回报为

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{1}$$

还有最重要的, 我们的模型, 或者说“策略”: π. 我们将其定义为给定状态下采取各种行动的概率的分布, 也即 $\pi(\cdot \mid s)$.

实际上, 我们定义的 $\langle S, A, P, r, \gamma \rangle$ 就是一个**马尔可夫决策过程**.

7.2 更多回报, 但是回报有多少?

我们希望得到更好的策略. 具体来说, 就是让该策略能够得到更大的总回报. 这种倾向反映在策略中, 也就是让一些能带来更大总回报的行动 a 的概率更高.

而在状态 s 上执行动作 a 时, 遵循策略 π 的总回报期望(**动作价值函数**)是

$$
\begin{aligned}
Q^{\pi}(s,a) &= \mathbb{E}(G_t \mid s_t = s, a_t = a) \\
&= \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k} \,\middle|\, s_t = s, a_t = a \right) \\
&= \mathbb{E}\left(R_t + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, s_t = s, a_t = a \right) \\
&= r(s,a) + \gamma \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, s_t = s, a_t = a \right)
\end{aligned} \tag{2}
$$

这里 s_{t+1} 是一个分布, 如果能得知在某个状态上执行策略 π 的总回报期望(**状态价值函数**), 我们还能继续分解上式末尾的 $\mathbb{E}(\cdot)$, 于是我们定义这个值

$$
V^{\pi}(s) = \sum_{a \in A} \pi(a \mid s) Q^{\pi}(s,a) \tag{3}
$$

正如刚刚说的, 我们可以继续变形 Q:

$$
\begin{aligned}
Q^{\pi}(s,a) &= r(s,a) + \gamma \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, s_t = s, a_t = a \right) \\
&= r(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^{\pi}(s')
\end{aligned} \tag{4}
$$

现在看看我们得到了什么

$$
Q^{\pi}(s,a) = r(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^{\pi}(s') \tag{5}
$$

$$
V^{\pi}(s) = \sum_{a \in A} \pi(a \mid s) Q^{\pi}(s,a) \tag{6}
$$

7.3 那么最优策略呢?

在我们推导了太多和一般策略有关的期望之后, 我们终于准备好研究一般策略的特例: **最优策略**.

事不宜迟, 我们定义一个策略 π^*, 使得 $\forall \pi, \forall s \in S, V^{\pi^*}(s) \geq V^{\pi}(s)$. 这是非常直观的最优策略, 因为无论在什么状态下, 其表现(回报期望)都比任意策略更好, 或至少一样.

根据该定义, 我们就有

$$
V^*(s) = \max_{\pi} V^{\pi}(s) \tag{7}
$$

和

$$
Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{8}
$$

一个好消息是, 既然最优策略是策略, 那他就符合我们上面推导出的一些结论, 比如 式 5, 于是我们有

$$
Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in S} P(s' \mid s,a) V^*(s') \tag{9}
$$

另一个好消息是, 既然最优策略是最优的, 他就会在每个状态上都采取 Q 最大的 a. 那么除非若干个行动有相同的 Q, $\pi^*(\cdot \mid s)$ 将会是 one-hot 的(只有一个行动的概率为 1, 其他都为 0), 否则其 V, 根据 式 6, 将并非最大. 无论是多个相同 Q 的行动分享 1 还是 one-hot, 都有

$$V^*(s) = \max_{a \in A} Q^*(s, a) \tag{10}$$

这其实是 式 6 的简化版本.

式 9 和 式 10 看起来是两个互相耦合的函数, 不过既然如此, 我们也可以把它们互相代入, 得到递归的形式.

式 9 $\rightarrow$ 式 10

$$V^*(s) = \max_{a \in A} \left(r(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) V^*(s') \right) \tag{11}$$

式 10 $\rightarrow$ 式 9

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s' \in S} P(s' \mid s, a) \max_{a' \in A} Q^*(s', a') \tag{12}$$

这两个式子就是**贝尔曼最优方程**.

7.4 如何求出它?

策略是抽象的, 听上去似乎不像是什么可以求解的对象, 尤其是当我们并没有讨论具体问题的时候. 但是通过上面的这些推演, 我们已经把抽象的策略变成了具体的数学对象 Q 和 V: 只要得到这两个函数中的一个(因为他们可以互相推导), 我们就事实上得到了最优策略, 因为我们可以选择 $\arg\max_a Q(s, a')$ 来让回报最大化.

下面我们不加证明地给出 Q-Learning 的迭代公式.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[R_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \Big] \tag{13}$$

当 Q-Learning 算法工作时, 我们先从一个随机初始化的 Q 和状态 s_t 开始, 将 Q 当作 Q^* 执行策略, 也即执行 $a_t = \arg\max_a Q(s_t, a_t)$. 每次得到环境的反馈 R_t, s_{t+1} 后, 就进行一次迭代, 然后重复这个过程.

虽然我并不打算细讲这个迭代公式的收敛性的具体证明过程, 但是我们还是有必要从直观上理解这个公式. 我们主要关注增量部分, 它包含

- α: 这是一个可调参数, 用于控制学习率, 这很好理解
- $R_t + \gamma \max_a Q(S_{t+1}, a)$: 利用环境给出的信息 R_t 展开 Q 的一项, 这是我们给出的新的 Q 的估计. 由于它正确地利用了环境给出的信息, 所以直觉上应该比原始的 Q 更接近最优
- $-Q(s_t, a_t)$: 减去原本的 Q 的估计, 得到增量部分, 这很好理解

但是这还不够, 事实上 Q-Learning 在选择动作时并没有完全按照估计的 Q 执行——算法执行初期我们希望得到更多环境的信息, 所以我们需要一些随机探索.

所以实际上 Q-Learning 采用的是 ε-贪心算法, 也就是有 ε 的概率随机选择 $a \in A$ 执行, 剩余的 $1 - \varepsilon$ 则贪心地选择 $\arg\max_a Q(s_t, a_t)$, 然后我们可以令 ε 随时间衰减, 就能达到我们"在算法运行初期增加随机探索"的目的了.

然而, Q-Learning 有个巨大的缺点: 他只能处理 S 和 A 都是有限集合的情形, 因为迭代公式只是在更新单个自变量的值. 这真是太坏了, 因为现实中的很多问题都是连续的, S 和 A 都是巨大的实向量空间, 这种情况下我们显然不能更新单个自变量对应的函数值来得到 Q, 我们要怎么办?

7.5 函数拟合? 有了

据我们所知, 神经网络很擅长这种拟合函数的工作. 但是要在这种问题上采用神经网络, 我们还需要得知网络的更新方式: 我们需要损失值.

损失值基本上就是模型输出和正确结果(至少是"更正确")之间的差异, 带着这种观点重新审视式 13, 我们会发现答案就在其增量部分. 只要简单地套上一个 MSE, 我们就找到了损失值

$$L = \mathrm{MSE}\Big(R_t + \gamma \max_a Q(s_{t+1}, a), Q(s_t, a_t)\Big) \tag{14}$$

我们已经很接近 **DQN**(Deep Q Network) 算法了, 还剩下一点实践中的考量

7.5.1 不要浪费样本

神经网络当然不能像 Q-Learning 那样采样一次就利用这次采样的数据迭代一次: 这样得到的数据并不"同分布", 于是神经网络只能学习到最近的数据; 每个样本只使用了一次, 效率太低.

所以, 我们会先连续地在环境中采样, 并把采样得到的 (s_t, a_t, R_t, s_{t+1}) 保存到一个**缓冲区**中, 等到缓冲区中有一定数量的样本后, 再进行训练.

7.5.2 "参考答案"也有网络的输出

我们这个神经网络的训练和监督学习的训练略有不同: MSE 的两项都包含神经网络本身的输出, 更新网络的时候目标也在改变, 这会导致训练不稳定.

所以我们干脆使用两套网络, 一套**目标网络**暂不更新, 用于计算 $\max_a Q(s_{t+1}, a)$, 另一套正常更新, 用于计算 $Q(s_t, a_t)$. 只有每隔 C 步, 才把目标网络和正常更新的网络进行同步, 从而保证训练的稳定性.

7.6 我想试试看

我已经跃跃欲试了, 有没有什么问题简单又适合 DQN 的让我做一做?

有的兄弟, 有的. 下面我们来看一个经典问题: 车杆问题(Cart-Pole Problem).

车杆问题(如 图 21)是一个经典控制问题, 其基本环境由一个可在水平轨道上左右移动的小车和一根铰接在小车上的直杆组成. 杆的初始状态略有倾斜, 因此会因重力而自然倾倒. 我们的目标是通过控制小车的左右移动, 使杆保持竖直平衡状态尽可能长的时间, 要求杆与竖直方向的夹角不超过特定阈值, 同时小车不能超出轨道边界.

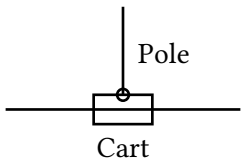

图 21　车杆问题示意图

7.6.1 定义环境

好消息是 Python 包 `gymnasium` 已经为我们实现了这个环境的代码, 我们可以直接调用.

```
from gymnasium.envs.classic_control import CartPoleEnv
from gymnasium.wrappers.common import TimeLimit
```

```
def get_env(render: False, max_step=-1):
    raw_env = CartPoleEnv(render_mode="human" if render else None)
    if max_step > 0:
        return TimeLimit(raw_env, max_episode_steps=max_step)
    return raw_env
```

这里不使用 `gym.make("CartPole-v1")` 的原因是其限制了最长步数为 500, 这样的步数仍然过短, 模型可能陷入局部最优解, 比如任小车以较慢的速度滑出有效区域.

这个环境的 S 和 A 如下表所示.

状态	取值区间
车的位置	$[-4.8, 4.8]$
车的速度	$\mathbb{R}$
杆的角度	$[-0.418, 0.418]$
杆末端的角速度	$\mathbb{R}$

表 2　车杆问题状态空间

动作	值
向左推车	0
向右推车	1

表 3　车杆问题动作空间

只要坚持一帧, 就能获得分数为 1 的奖励.

7.6.2 设计 Q 网络

根据环境的 S 和 A, 我们需要设计一个接收一个四维向量, 并输出一个二维向量的网络. 我这里给出一个例子.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNet(nn.Module):
    def __init__(self, state_size=4, action_size=2):
        super(QNet, self).__init__()

        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, mode='fan_in',
nonlinearity='relu')
                nn.init.constant_(m.bias, 0)
```

```
    def forward(self, state):
        if state.dim() == 1:
            state = state.unsqueeze(0)

        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        q_values = self.fc3(x)

        return q_values
```

这是一个具有两个隐藏层的网络, 并使用 ReLU 作为激活函数, 还使用了 He 初始化, 优化了初期的训练.

7.6.3 样本缓冲区

我们用 `collections.deque` 做一个简单的样本缓冲区, 可以向里面存入样本, 然后随机地取出.

```
import collections
import random
import numpy as np

class SampleBuffer:
    def __init__(self, max_size):
        self.buffer = collections.deque(maxlen=max_size)

    def add(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        samples = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*samples)
        return np.array(state), action, reward, np.array(next_state), done

    def size(self):
        return len(self.buffer)
```

7.6.4 DQN 算法

正如我们前面所说的, 我们实现 DQN 算法的更新算法和ε-贪心策略. 有一点不同的是, 我们的环境会因为某些原因终止, 比如杆的角度或者小车位置超出范围, 到达最大时间等. 当环境终止时, 要把 式 14 MSE 中第一项(也即下面代码的 `q_targets`) 中对未来的估计部分变为 0, 因为环境已经终止, 未来不会有任何回报了.

```
class DQN:
    def __init__(self, state_dim, action_dim, learning_rate, gamma, epsilon,
target_update_freq, device):
        self.action_dim = action_dim

        self.q_net = QNet(state_dim, action_dim).to(device)
        self.target_q_net = QNet(state_dim, action_dim).to(device)

        self.optimizer = torch.optim.Adam(
            self.q_net.parameters(), lr=learning_rate)
        self.gamma = gamma
        self.epsilon = epsilon
```

```
        self.target_update = target_update_freq
        self.update_count = 0
        self.device = device

    def take_action(self, state):
        # epsilon-greedy
        if np.random.random() < self.epsilon:
            action = np.random.randint(self.action_dim)
        else:
            state = torch.tensor([state]).to(self.device)
            action = self.q_net(state).argmax().item()
        return action

    def update(self, transition_dict):
        states = torch.tensor(transition_dict['states']).to(self.device)
        actions = torch.tensor(
            transition_dict['actions']).view(-1, 1).to(self.device)
        rewards = torch.tensor(
            transition_dict['rewards']).view(-1, 1).to(self.device)
        next_states = torch.tensor(
            transition_dict['next_states']).to(self.device)
        dones = torch.tensor(
            transition_dict['dones']).view(-1, 1).to(self.device)

        # Q(s_t, a_t)
        q_values = self.q_net(states).gather(1, actions)

        # max_a Q(s_(t + 1), a)
        max_next_q_values = self.target_q_net(
            next_states).max(1)[0].view(-1, 1)

        # r(s, t) + gamma max_a Q(s_(t + 1), a), mul (1-done) for obvious reason
        q_targets = rewards + self.gamma * max_next_q_values * (1 - dones)

        loss = F.mse_loss(q_targets, q_values)

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # update target network
        if self.update_count % self.target_update == 0:
            self.target_q_net.load_state_dict(self.q_net.state_dict())
        self.update_count += 1
```

7.6.5 开始训练

设置参数, 初始化模型, 然后根据我们前面所述的策略启动训练.

```
import tqdm.notebook as tqdm

device = torch.device(
    "cuda") if torch.cuda.is_available() else torch.device("cpu")

lr = 2e-3
num_episodes = 500
```

```
gamma = 0.98
epsilon = 0.01
target_update = 10
buffer_size = 10000
min_buffer_size = 500
batch_size = 64

env = get_env(render=False, max_step=2000)
replay_buffer = SampleBuffer(buffer_size)
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n
agent = DQN(state_dim, action_dim, lr, gamma, epsilon,
            target_update, device)

return_list = []
for i in range(10):
    with tqdm.tqdm(range(int(num_episodes / 10)), desc='Iteration %d' % i) as pbar:
        for i_episode in pbar:
            episode_return = 0
            state, _ = env.reset()
            done = False
            while not done:
                action = agent.take_action(state)
                next_state, reward, terminated, truncated, _ = env.step(action)
                done = terminated or truncated
                replay_buffer.add(state, action, reward, next_state, 1 if done else
0)
                state = next_state
                episode_return += reward

                # train after there are enough samples
                if replay_buffer.size() > min_buffer_size:
                    b_s, b_a, b_r, b_ns, b_d = replay_buffer.sample(batch_size)
                    transition_dict = {
                        'states': b_s,
                        'actions': b_a,
                        'next_states': b_ns,
                        'rewards': b_r,
                        'dones': b_d
                    }
                    agent.update(transition_dict)
            return_list.append(episode_return)
            if (i_episode + 1) % 10 == 0:
                pbar.set_postfix({
                    'episode':
                    '%d' % (num_episodes / 10 * i + i_episode + 1),
                    'return':
                    '%.3f' % np.mean(return_list[-10:])
                })
```

7.6.6 观察结果

新建一个有可视化界面的环境, 然后看看模型的表现吧!

```
env = get_env(render=True)
```

```
state, _ = env.reset()
done = False
while not done:
    action = agent.take_action(state)
    env.render()
    next_state, reward, terminated, truncated, _ = env.step(action)
    done = terminated or truncated
    state = next_state
```

不出意外的话, 你的模型应该能很好地控制住杆, 直到永远.

如果不幸没有, 请再仔细检查代码有没有错误. 此外由于我并没有设置任何 `seed`, 所以这种情况也是完全有可能发生的, 可以再尝试几次, 或者尝试修改超参数.

无论如何, it works on my machine.

7.7 好像还缺点什么

我们批评过 Q-Learning 只能学习离散状态空间中的 Q 函数, 但现在我们的 DQN 也不能给出连续的动作.

是的, 所以研究者们还提出了基于"策略梯度"的算法(DQN 是基于值函数的), 比如 REINFORCE, 这种算法可以预测连续的动作. 除此之外, 还有混合两种思想的算法, 比如 Actor-Critic, PPO, DDPG 等. 但是这些算法都不在本文的计划范围之内了.

至此, 我们已经完成了对强化学习基础概念和两种入门算法的介绍. 从最初的马尔可夫决策过程, 到价值函数, 策略函数的概念, 再到 Q-Learning 和 DQN 算法的实现, 我们循序渐进地窥见了强化学习的一点思想.

不管怎样, 强化学习是一个广阔而深刻的领域, 本文也将仅仅止步与其思想和入门算法的介绍. 上面提到的其他算法并不在本文的计划内容范围内. 若有兴趣, 可以自行了解.

希望你有所收获.

2025-02-05

florr.io 中的合成与概率

在 WSL 中创建一个 alias, 直接使用 Windows 文件资源管理器打开目录或文件.

florr.io 是一款~~开放世界冒险游戏~~, 玩家可以在自己控制的花朵(flower)上装备各种花瓣(petal)来与地图中昆虫或者其他物体战斗.

花瓣有不同种类, 相同的种类也有等级高低之分, 高等级的花瓣可以通过击败高等级的昆虫, 并拾取其掉落物获得, 也可以通过五个次一级的同种花瓣合成获得.

8.1 花瓣合成的机制

花瓣合成的具体机制如下:

1. 一次合成是否成功服从伯努利分布, 对于大多数花瓣来说, 各等级合成的成功概率为

合成等级	成功率
普通合成罕见	64%
罕见合成稀有	32%
稀有合成史诗	16%
史诗合成传奇	8%
传奇合成神话	4%
神话合成究极	2%
究极合成超级	1%

表 4　花瓣合成成功率表

2. 如果合成失败, 将会随机销毁一个或数个花瓣, 销毁的花瓣数服从 $U(1,4)$.

为了后续讨论的方便, 这里我们假设每次合成都是独立的, 也即不存在“保底”等规则.

8.2 我需要多少次级花瓣

有些时候, 我们想要知道需要多少次级的花瓣才能有较大概率(比如 95%)获得至少一个高级花瓣.

这里我们设一次合成成功率为 p, 为了有 k 的概率获得至少一个高级花瓣, 我们有

$$1-k=(1-p)^{m_k}$$

$$\Rightarrow m_k=\frac{\ln(1-k)}{\ln(1-p)}$$

其中 m_k 是预期的合成次数.

这 m_k 次中包含了 m_k-1 次合成失败和 1 次合成成功, 对于每次合成失败, 预期的损耗为 $\frac{5}{2}$, 所以

$$\begin{aligned} n_k &= \frac{5}{2}(m_k - 1) + 5 \\ &= \frac{5}{2}\left(\frac{\ln(1-k)}{\ln(1-p)} + 1\right) \end{aligned}$$

其中 n_k 是需要的次级花瓣数.

根据此公式, 我们得到如下表格

合成等级	成功率	$n_{0.95}$	$n_{0.99}$
普通合成罕见	64%	9.83	13.77
罕见合成稀有	32%	21.92	32.35
稀有合成史诗	16%	45.45	68.53
史诗合成传奇	8%	92.32	140.58
传奇合成神话	4%	185.96	284.53
神话合成究极	2%	373.21	572.37
究极合成超级	1%	747.68	1148.03

表 5　花瓣合成所需次级花瓣数表