获取Polar数据流程2

这个阶段接着获取Polar数据流程1的下一个阶段。使用真机来发送数据流，再通过LSL获取并记录数据的流程。

0. 目标与范围

目标

在不依赖任何供应商封闭软件的前提下，建立一条稳定、可验证、可扩展的数据链路，用于将移动端产生的心率/事件标注等数据，经由 UDP 发送到 Mac 端，再发布为两路 LSL 流并由 LabRecorder 录制到单一 `.xdf` 文件。该链路将作为后续接入 Polar Verity（PPI/HR）与 Polar H10（ECG）的公共骨干。

范围

本阶段仅实现：

- UDP→LSL 桥接器（Python）
- 两路 LSL 流：数据流 `PB_UDP[_TEST]` 与标记流 `PB_MARKERS[_TEST]`
- 会话级命名与旁路日志（ `.jsonl` ）
- 固化的操作顺序（SOP）与“预检 + 录后 QA”自检机制

---

1. 系统架构与时序

链路

iPhone App → UDP 发包 → 局域网 → `udp_to_lsl.py` （Mac） → 在本机发布两路 LSL → LabRecorder 发现并录制 → `.xdf` 文件

在这个链路中，手机 App 是**数据源**之一，把心跳或将来的 Verity/H10 数据序列化为 JSON，经 UDP 发给桥接器（ `udp_to_lsl.py` ），同时把把“基线/刺激/干预/结束”以 `{"type":"marker",...}` 形式发给桥接器。

`udp_to_lsl.py` 是桥接器。手机端流出的数据（JSON格式）经由UDP传入桥接器，再转化为两个LSL流。字符串型数据流（暂命名为 `PB_UDP` ），包含“心跳”“PPI/ECG JSON”等字符串。字符串型标记流（暂命名为 `PB_MARKERS` ）写入 `{"type":"marker","label":"..."}` 的标签。

LabRecorder 是**录制器**，在网络上“发现”桥接器发布的 LSL 流，并1. - 把它们采集到同一份 `.xdf` 文件中，时间轴对齐。

时序要点

1. 启动桥接器（创建两路 LSL 流）。
2. LabRecorder 发现并勾选这两路流，点击 Start。
3. 手机端开始发送；在需要的时刻发送标记（`baseline_start / stim_start / stim_end` 等）。
4. 手机端停止发送。
5. LabRecorder Stop，写出 `.xdf`。
6. 录后 QA 校验通过，归档。

时间基准

- 桥接器以 `pylsl.local_clock()` 作为样本时间戳来源。
- 移动端可附带自身时间戳，但入 LSL 的主时间轴以 Mac 为准，保证与 EEG/SCR 端统一时钟基准。

---

2. 环境与依赖

- macOS 13+（Apple Silicon 已验证）
- 注册真机，这个环节遇到了“超出3个iphone设备”的问题，没找到解决方法，最后通过另一个账号解决该问题。
- Python 3.10/3.11（建议统一使用同一解释器路径）
- 依赖包：
  - `pylsl`
  - `pyxdf`（录后 QA 与检查脚本用）
- LabRecorder（已解决签名与隔离标记--即文件损坏--问题）
- 局域网连通（iPhone 与 Mac 同网段），UDP 9001 端口开放（macOS 防火墙允许 Python 入站）。

---

3. 核心组件

3.1 UDP→LSL 桥接器：`udp_to_lsl.py`

功能

`udp_to_lsl.py` 是**桥接器（bridge）**，它从 UDP 端口接收文本（JSON），将其**路由**（转化）为两个 LSL 流并发布到本机局域网。该脚本创建并持续发布的 LSL outlets。只要该脚本在运行，

LabRecorder 的“Update”刷新后依然会看到同样的两个流：`PB_UDP`（数据流，string）和`PB_MARKERS`（标记流，string）。

- 监听 `0.0.0.0:9001`（可在脚本顶部配置区改）
- 将收到的文本 JSON 按类型路由：
  - `{"type":"marker","label":"..."}` → 推送到 `PB_MARKERS[_TEST]`（string）
  - 其它文本 → 推送到 `PB_UDP[_TEST]`（string）
- 每条入站 UDP 同时写入旁路日志 `<SESSION>.jsonl`。该JSON一个按时间戳命名的会话 ID，包含主机接收时间 `ts_host` 与原始文本 `raw`,旁路日志会按会话命名保存。也可以手工设定，便于批量实验归档。
- 维护数据/标记的计数器，周期性打印摘要。旁路日志：所有入站 UDP 文本都会落到 `~/lsl_logs/<SESSION>.jsonl`，便于排查“我按了按钮但 XDF 里没看到”的问题。
- 对空标记自动替换为 `"unknown"`，避免空字符串进入标记流

命名与会话

- 流名：`PB_UDP[_TEST]`、`PB_MARKERS[_TEST]`。开发联调阶段建议保留 `_TEST`，稳定后可改为空字符串回到正式名。
- `SESSION`：默认用当前时间戳生成，可手动设定，旁路日志按会话命名。

可靠性约束

- UDP 单向，不做重传；在 1 Hz 心跳场景可接受偶发丢一两包。
- 通过旁路日志与计数摘要对照 XDF 样本数，快速判断是否存在范围性丢包或操作顺序错误。

3.2 iPhone App（SwiftUI）

当前实现

未来Polar检测的心率数据都需要从此App获取，它本质上就是数据源获取和发送器。

- 在前一版的基础上又修改了UI，加入了几个标注按钮（基线开始、刺激开始、刺激结束、干预开始、干预结束）。这些按钮可以为数据流加标记。
- 把目标IP (default host) 更新为MAC实际的局域网 IP。检查MAC的IP地址用 `ipconfig getifaddr en0`。在真机上 `127.0.0.1` 是手机自身，不会到达 Mac。改成本机局域网 IP 才能送到 Mac。
- 三类发送：开始每秒发送（1 Hz 心跳 JSON）、停止、发送一次
- 三个标记按钮：`baseline_start / stim_start / stim_end`
- 成功发送后在 Xcode 控制台打印 `TICK` 或 `MARKER ...`，便于现场排查

> 后续接入 Polar SDK 后，心跳 JSON 将替换/并行为 PPI/HR 或 ECG 数据 JSON。

3.3 LabRecorder

设置

- 录制前选择目录，手动命名文件（建议以会话命名）
- Update/Refresh 后只勾选本次目标流（`PB_UDP[_TEST]` 与 `PB_MARKERS[_TEST]`）
- Start/Stop 的时序必须覆盖“手机开始发送”与“手机停止发送”之间的区间

---

4. 标准作业流程（SOP）

准备

1. 关闭可能占用 9001 的其他进程（`lsof -nP -iUDP:9001` 应只见一个 Python）。
2. 关闭 VPN/代理，如有多条 `utun` 接口导致 LSL 发现异常，先断开再试。
3. iPhone 与 Mac 同一局域网，手机端设置目标 IP 为 Mac 的内网 IPv4，端口为 9001。

执行

执行流程为：首先打开桥接器，准备好数据输入和输出之间的通道。接下来打开接受端，准备数据流入。当通道、接受都准备好以后，再打开数据输入端，也就是数据源，开始传播数据。所以严格按这个顺序操作（关键是**先开桥接器，再开 LabRecorder，再开始发送**）：

1. 运行 `udp_to_lsl.py`，确认终端打印两路 LSL 已创建，且随后每秒出现 `[DATA #n]`。
2. 打开 LabRecorder，选新目录与文件名；Update 后只勾选两路目标流；点击 Start。
3. 在手机端点击“开始每秒发送”，随后按需点击三个标记按钮。
4. 停止发送；LabRecorder Stop。
5. 运行检查脚本（见下一节）验证通过后归档。

> 两个常见“坑”：
> • 在 LabRecorder **停止**之后才开始“开始每秒发送”，那录出来自然只有 0 或 1。
> • 打开了旧文件检查，`check_xdf.py` 的首行会打印你正在检查的路径，请确认时间戳。

验收口径

- `PB_UDP[_TEST]` 的 `samples ≈ span 秒数`（1 Hz 心跳时约等于录制时长）
- `PB_MARKERS[_TEST]` 至少 3 条；`Markers:` 列表包含刚才按下的标签
- 旁路日志 `.jsonl` 存在并可追溯

---

5. 质量保障：预检与录后 QA

5.1 预检（preflight）

- 在录制前运行“预检脚本”，对目标流进行 10 秒内发现与拉样本测试。
- 通过标准：能发现目标流名，能在 2 秒超时内拉到至少 1 条样本。
- 目的：提前排除“LSL 未发现/取样失败/端口冲突/VPN 干扰”等。

5.2 录后 QA（qa_check）

- 对刚保存的 `.xdf` 执行：
  - 数据流与标记流是否存在
  - 数据流时长是否达到最小阈值（例如 ≥8 秒）
  - 标记条数是否达到最小阈值（例如 ≥2）
  - 数据样本密度与时长是否明显不匹配（告警）

> 使用 `check_xdf.py` 经修正后支持按 `type=Markers` 或名称包含 `markers` 自动识别标记流，建议保留该逻辑。

---

6. 常见故障与快速定位

现象	高概率原因	定位与处置
`.xdf` 中 `PB UDP=1, span=0s`	录制窗口未覆盖手机发送窗口；检查了旧文件	严格按 SOP 重录；使用新目录与文件名； `check_xdf.py` 首行打印路径确认；先 Start 再“开始每秒发送”，再 Stop
`Markers` 列表缺失或有空字符串	录制时桥接器旧版本未做空标签替换；脚本仅按固定名搜索	保持桥接器“空标签→unknown”； `check_xdf.py` 用“按 type 或名称包含 markers”匹配
`peek_lsl.py` 找不到流	LSL 发现超时太短；VPN/多 utun 干扰；未运行桥接器/未在同网段	将发现超时调到 10 秒；断开 VPN；确保 iPhone 与 Mac 同网段；确认桥接器终端每秒打印 `[DATA]`
桥接器端口绑定失败	端口被占用	`lsof -nP -iUDP:9001` 查出占用进程后关闭
LabRecorder 看到多条同名流	开了多个桥接器实例或旧实例未退出	只保留一个桥接器；开发期用 `_TEST` 后缀隔离
样本数与桥接器计数差异明显	UDP 丢包异常或 LabRecorder 勾选错误	优先检查是否只勾选了两路目标流；如需更稳可把关键数据转 WebSocket/TCP（本阶段不做）

---

7. 设计决策与取舍

- **UDP**：选择原因是实现简单、时延低、无连接开销，适合 1–10 Hz 的状态/心跳/标记类数据。缺点是无重传。为此在桥接器端做计数与旁路日志，用 QA 弥补可靠性。
- **LSL 时间戳以 Mac 为准**：保证与 EEG/SCR 等桌面设备的同步。移动端时间戳可并行记录，做离线比对与校正。
- **两路流（数据与标记）**：简化分析；LabRecorder 与后处理均有成熟惯例。
- **会话命名与 `_TEST` 后缀**：避免旧流/旧文件混淆；当稳定后去掉 `_TEST` 回到正式名。
- **旁路 `.jsonl`**：作为取证与排错依据，必要时可与 `.xdf` 对齐重建样本级事件序列。

附1：新添加和有修改的代码清单

> 代码已在本地实现并验证通过，这里只列新添加和修改的代码。

`udp_to_lsl.py`

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# —— 配置区：只改这里 ————————————————————————————————
from pathlib import Path
import time

CONFIG = {
# UDP 监听地址与端口（iPhone/发包端要把目标指向本机IP:PORT）
"HOST": "0.0.0.0",
"PORT": 9001,

# 会话ID：默认用时间戳。若想手工固定，改成 "S20250820-01" 这类字符串
"SESSION": time.strftime("S%Y%m%d-%H%M%S"),

# LSL 流名后缀：开发期建议用 "_TEST"，稳定后可改成空字符串 ""
"NAME_SUFFIX": "_TEST",

# 旁路日志目录（逐条 UDP 入站会落到这里的 .jsonl）
"LOGDIR": str(Path.home() / "lsl_logs"),

# 控制台统计摘要的间隔秒数
"SUMMARY_EVERY": 5,
}
# ——————————————————————————————————————————————————

```

```
import socket, json, uuid, os
from pylsl import StreamInfo, StreamOutlet, local_clock

def _name(base: str) -> str:
    suf = CONFIG["NAME_SUFFIX"] or ""
    return f"{base}{suf}"

def _make_outlet(name: str, stype: str, source_id: str, channel_format: str = "string"):
    info = StreamInfo(name, stype, 1, 0.0, channel_format, source_id)
    desc = info.desc()
    desc.append_child_value("impl", "udp_to_lsl_v2")
    desc.append_child_value("session", CONFIG["SESSION"])
    desc.append_child_value("created_at", time.strftime("%Y-%m-%dT%H:%M:%S"))
    return info, StreamOutlet(info, chunk_size=0, max_buffered=360)

def main():
    # 准备日志
    Path(CONFIG["LOGDIR"]).mkdir(parents=True, exist_ok=True)
    log_path = Path(CONFIG["LOGDIR"]) / f"{CONFIG['SESSION']}.jsonl"
    logf = open(log_path, "a", buffering=1, encoding="utf-8")

    # 绑定 UDP
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        sock.bind((CONFIG["HOST"], CONFIG["PORT"]))
    except OSError as e:
        print(f"[FATAL] UDP {CONFIG['HOST']}:{CONFIG['PORT']} bind failed: {e}")
        return

    sock.setblocking(True)
    # 唯一 source_id
    sid_data = f"pb_udp_{CONFIG['SESSION']}_{uuid.uuid4().hex[:8]}"
    sid_mark = f"pb_markers_{CONFIG['SESSION']}_{uuid.uuid4().hex[:8]}"

    # 两路 LSL 流
    name_data = _name("PB_UDP")
    name_mark = _name("PB_MARKERS")
    info_data, outlet_data = _make_outlet(name_data, "udp_text", sid_data)
    info_mark, outlet_mark = _make_outlet(name_mark, "Markers", sid_mark)

    print(f"[udp_to_lsl] session={CONFIG['SESSION']}")
    print(f"[udp_to_lsl] listening on {CONFIG['HOST']}:{CONFIG['PORT']}")
```

```
    print(f"[udp_to_lsl] LSL outlets: {name_data} (sid={sid_data}), {name_mark} (sid={sid_mark})")
    print(f"[udp_to_lsl] log file: {log_path}")

    count_data = 0
    count_mark = 0
    t0 = time.time()

    while True:
        data, addr = sock.recvfrom(65535)
        ts = local_clock()

        try:
            text = data.decode("utf-8", errors="ignore").strip()
        except Exception:
            text = f"<{len(data)} bytes>"

        # 旁路日志：每条 UDP 入站都写盘
        logf.write(json.dumps({"ts_host": time.time(), "remote": addr, "raw": text}) + "\n")

        # 路由：marker 单独走标记流，其它都进数据流
        routed = False
        try:
            obj = json.loads(text)
            if isinstance(obj, dict) and obj.get("type") == "marker":
                raw_label = obj.get("label", "")
                label = raw_label if isinstance(raw_label, str) and raw_label.strip() else "unknown"
                outlet_mark.push_sample([label], timestamp=ts)
                count_mark += 1
                print(f"[MARK #{count_mark}] {addr} -> {label}")
                routed = True
        except Exception:
            pass

        if not routed:
            outlet_data.push_sample([text], timestamp=ts)
            count_data += 1
            print(f"[DATA #{count_data}] {addr} -> {text}")

        # 周期性摘要
        now = time.time()
        if now - t0 >= CONFIG["SUMMARY_EVERY"]:
            print(f"[SUMMARY] data={count_data}, markers={count_mark}, elapsed={int(now - t0)}s")
```

```
            t0 = now

if __name__ == "__main__":
    main()
```

该程序的功能包括：

- 监听 `0.0.0.0:9001`
- 发布 `PB_UDP[_TEST]` 与 `PB_MARKERS[_TEST]`
- 写 `~/lsl_logs/<SESSION>.jsonl`
- 控制台打印 `[DATA #n]`、`[MARK #m]` 与周期性 `[SUMMARY]`

peek_lsl.py

```python
# peek_lsl.py
from pylsl import StreamInlet, resolve_byprop
print("Resolving PB_UDP ...")
streams = resolve_byprop('name','PB_UDP', timeout=5)
if not streams:
    print("PB_UDP not found")
    raise SystemExit(1)
inlet = StreamInlet(streams[0], max_buflen=60)
print("Connected. Reading 10 samples:")
for i in range(10):
    sample, ts = inlet.pull_sample(timeout=2)
    if sample is None:
        print("timeout")
    else:
        print(i+1, ts, sample[0])
```

该程序功能包括：

- 验证 LSL 发现与取样（独立于 LabRecorder）

check_xdf.py

```python
import sys, os
from pathlib import Path
from collections import Counter

try:
    # 可选：用于弹出文件选择框；若缺失会自动回退
    from tkinter import Tk, filedialog  # type: ignore
except Exception:
```

```
    Tk = None

from pyxdf import load_xdf

def pick_file_dialog():
    if Tk is None:
        return None
    root = Tk()
    root.withdraw()
    root.update()
    path = filedialog.askopenfilename(
        title="Select XDF file",
        filetypes=[("XDF files", "*.xdf"), ("All files", "*.*")]
    )
    root.destroy()
    return path or None

def resolve_path():
    # 1) 命令行参数
    if len(sys.argv) > 1:
        return sys.argv[1]
    # 2) 环境变量
    env = os.getenv("XDF_PATH")
    if env:
        return env
    # 3) 图形文件对话框
    return pick_file_dialog()

# 寻找“标记”流：优先按 type=Markers，其次按 name 含 markers
def is_marker_stream(s):
    name = s["info"]["name"][0] if s["info"]["name"] else ""
    typ  = s["info"]["type"][0] if s["info"]["type"] else ""
    return (typ and typ.lower() == "markers") or ("markers" in
name.lower())

def main():
    path = resolve_path()
    if not path:
        print("请这样使用：\n"
              "  A) 终端：python check_xdf.py /path/to/file.xdf\n"
              "  B) 终端：XDF_PATH=/path/to/file.xdf python check_xdf.py\n"
              "  C) 直接运行：弹出文件对话框（若 Tkinter 不可用，则请用 A 或 B）")
        sys.exit(2)

    path = str(Path(path).expanduser())
    streams, hdr = load_xdf(path)
```

```
54
55      print(f"File: {path}")
56      for s in streams:
57          name = s["info"]["name"][0]
58          typ  = s["info"]["type"][0] if s["info"]["type"] else ""
59          n    = len(s["time_series"])
60          if n:
61              t0 = s["time_stamps"][0]
62              t1 = s["time_stamps"][-1]
63              dur = t1 - t0
64              print(f"{name:20s} | type={typ:10s} | samples={n:6d} | span=
    {dur:7.2f}s")
65          else:
66              print(f"{name:20s} | type={typ:10s} | samples={n:6d}")
67
68      # 打印 markers 内容（若存在）
69      marker_streams = [s for s in streams if is_marker_stream(s)]
70      if marker_streams:
71          s = marker_streams[0]
72          labels = [row[0] for row in s["time_series"]]
73          # 打印全部标签与出现次数
74          print("Markers:", labels)
75          from collections import Counter
76          print("Marker counts:", Counter(labels))
77      else:
78          print("No marker stream found by type/name.")
79
80  if __name__ == "__main__":
81      main()
```

该程序功能包括：

- 打印每条流的样本数与跨度；输出 `Markers:` 列表与计数
- 已修正为按 `type=Markers` 或名称包含 `markers` 识别标记流

qa_check.py

```
1   """
2   qa_check.py
3   用途：对一段刚录完的 XDF 文件做检查，按几条简单的必须满足的规则判断这次录制是不是合格。
    任意一条不达标，就当这次录制作废，立刻重录。该脚本目的只有一个：快速发现“录错文件、勾错
    流、窗口没覆盖、忘了按按钮”这类人为错误，省得到分析阶段才发现白忙活一场。
4
5   由于本流程比较繁琐，可能会不住细节、顺序容易错、流名容易混、目录容易点错。这个脚本帮你机
    械地检查三件事：录了多久、有没有标记、标记够不够。只要任何一条不过关，它就告诉你“这次无
```

```
效”。接入 Verity/H10，这个脚本照样能用，对于高阶校验（比如波形质量、心率区间合理性）可以在后续再加更“细”的检查脚本。

规则死在脚本顶部，可以在脚本顶部“配置区”调阈值。脚本会打印：
- 文件路径
- 找到的每条流的名称、类型、样本数、时间跨度
- 标记列表与计数
- 结论：PASS / NOT PASS，并列出失败原因或告警。具体来说：

如果是 PASS，这次录制可用。
如果是 NOT PASS，输出会告诉你具体哪里不达标（比如没找到标记流、时长太短、标记太少）。直接按提示重录即可。
如果有 [WARN] 样本密度偏低，一般是手机发送频率和你设定的期望不一致（比如心跳本来不是 1 Hz），你可以：
要么把脚本顶部 EXPECTED_RATE_HZ 改成实际频率；
要么先不管这条告警（不影响 PASS/NOT PASS）。
"""
import sys

from collections import Counter

# “无命令行参数也能在 IDE 直接运行”加入可选的文件对话框支持
# 若系统无 Tkinter，可降级为从控制台输入路径
try:
    import os
    from tkinter import Tk, filedialog  # 可能在某些 Python 发行版中不可用
except Exception:
    Tk = None
    filedialog = None

def pick_file_dialog():
    """[新增] 弹出文件选择框选择 .xdf 文件；若 Tk 不可用则返回 None"""
    if Tk is None or filedialog is None:
        return None
    root = Tk()
    root.withdraw()
    root.update()
    path = filedialog.askopenfilename(
        title="Select XDF file",
        filetypes=[("XDF files", "*.xdf"), ("All files", "*.*")]
    )
    root.destroy()
    return path or None

# ── 配置区：按需要调整 ──────────────────────────────────────────
```

```python
MIN_DURATION_SEC    = 8      # 数据流最少录制秒数（防止“刚点开始就停止”的空文件）
MIN_MARKERS         = 2      # 最少标记条数（例如 baseline_start 与 stim_start）
EXPECTED_RATE_HZ    = 1.0    # 期望的数据流频率（当前心跳 1Hz）。若不想做密度校验，设为 0
DENSITY_FACTOR_MIN  = 0.7    # 密度阈值：样本数至少达到 0.7 * （时长 * 期望频率），否则告警
# 名称识别优先级（可不改）：先按 type，再按 name 子串
DATA_NAME_HINTS     = ["udp"]        # 名称里含有这些子串的，优先当作“数据流”候选
MARKER_NAME_HINTS   = ["markers"]    # 名称里含有这些子串的，优先当作“标记流”候选
# ——————————————————————————————————————————————————————————————

def _name(s): return s["info"]["name"][0] if s["info"]["name"] else ""
def _type(s): return s["info"]["type"][0] if s["info"]["type"] else ""

def _span_seconds(s):
    ts = s.get("time_stamps", None)
    if ts is None:
        return 0.0
    try:
        n = len(ts)
    except Exception:
        n = 0
    if n == 0:
        return 0.0
    return float(ts[n - 1] - ts[0])

def _pick_streams(streams):
    """
    返回 (data_stream, marker_stream)
    选择策略：
      - 标记流：优先 type=Markers；否则 name 包含 'markers'。
      - 数据流：优先 type!=Markers 且 name 包含 'udp'；否则在非标记流中选跨度最长的一条。
    """
    data = None
    mark = None

    # 先找标记流
    candidates_mark = []
    for s in streams:
        t = (_type(s) or "").lower()
        n = (_name(s) or "").lower()
        if t == "markers" or any(h in n for h in MARKER_NAME_HINTS):
            candidates_mark.append(s)
    if candidates_mark:
```

```
            # 如果有多条，取跨度最长的那条
            mark = max(candidates_mark, key=_span_seconds)

        # 再找数据流（排除已经当作标记的）
        candidates_data = []
        for s in streams:
            if s is mark:
                continue
            t = (_type(s) or "").lower()
            n = (_name(s) or "").lower()
            # 排除明显的标记类型
            if t == "markers":
                continue
            score = 0
            if any(h in n for h in DATA_NAME_HINTS):
                score += 10
            # 用跨度作为次要指标
            candidates_data.append((score, _span_seconds(s), s))
        if candidates_data:
            candidates_data.sort(key=lambda x: (x[0], x[1]), reverse=True)
            data = candidates_data[0][2]

        return data, mark

    # 从一行 time_series 中提取标签，兼容 numpy.ndarray/bytes/列表/元组/字符串
    def _extract_label(row):
        try:
            import numpy as np  # 只在需要时导入
            if isinstance(row, np.ndarray):
                row = row.tolist()
        except Exception:
            pass
        if isinstance(row, (list, tuple)):
            v = row[0] if row else ""
        else:
            v = row
        if isinstance(v, bytes):
            try:
                v = v.decode("utf-8", errors="ignore")
            except Exception:
                v = str(v)
        else:
            v = str(v)
        return v

    def main():
```

```
    # 参数处理：支持三种方式获取路径
    # 1) 命令行参数；2) 文件对话框；3) 控制台输入
    if len(sys.argv) >= 2:
        path = sys.argv[1]
    else:
        path = pick_file_dialog()
        if not path:
            try:
                path = input("请输入 XDF 文件路径：").strip('"').strip("'").strip()
            except EOFError:
                path = ""

    # [新增] —— 基本有效性检查
    if not path:
        print("未提供有效的 .xdf 路径。")
        sys.exit(1)
    if 'os' in globals() and not os.path.isfile(path):
        print(f"文件不存在：{path}")
        sys.exit(1)


    try:
        import pyxdf
    except ImportError:
        print("缺少依赖：pyxdf。请先执行：pip install pyxdf")
        sys.exit(1)

    try:
        streams, hdr = pyxdf.load_xdf(path)
    except Exception as e:
        print(f"[ERROR] 无法读取 XDF：{e}")
        sys.exit(1)

    # 打印所有流的基本信息
    print(f"[FILE] {path}")
    if not streams:
        print("[FAIL] 文件中没有任何流")
        sys.exit(1)

    print("[STREAMS]")
    for s in streams:
        n = _name(s)
        t = _type(s)
        cnt = len(s["time_series"])
        sp = _span_seconds(s)
```

```
        print(f"  - name='{n}' | type='{t or ''}' | samples={cnt} | span={sp:.2f}s")

    data, mark = _pick_streams(streams)

    all_pass = True

    # 规则 1：必须同时存在数据流与标记流
    if data is None:
        print("[FAIL] 未找到“数据流”（例如名字里含 udp 的那条）")
        all_pass = False
    if mark is None:
        print("[FAIL] 未找到“标记流”（type=Markers 或名字里含 markers）")
        all_pass = False

    # 若缺流，直接给出结论
    if not all_pass:
        print("[RESULT] NOT PASS")
        sys.exit(1)

    # 统计关键指标
    n_data = len(data["time_series"])
    n_mark = len(mark["time_series"])
    dur_data = _span_seconds(data)
    dur_mark = _span_seconds(mark)

    # 打印标记详情
    labels = [_extract_label(row) for row in mark["time_series"]]
    print(f"[DATA ] samples={n_data} | span={dur_data:.2f}s | name='{_name(data)}'")
    print(f"[MARK ] samples={n_mark} | span={dur_mark:.2f}s | name='{_name(mark)}'")
    print(f"[LABEL] {labels}")
    print(f"[COUNT] {Counter(labels)}")

    # 规则 2：数据流最短时长
    if dur_data < MIN_DURATION_SEC:
        print(f"[FAIL] 数据时长 {dur_data:.2f}s < 最小要求 {MIN_DURATION_SEC}s")
        all_pass = False

    # 规则 3：最少标记数
    if n_mark < MIN_MARKERS:
        print(f"[FAIL] 标记条数 {n_mark} < 最小要求 {MIN_MARKERS}")
        all_pass = False

```

```
    # 规则 4：样本密度（可选，默认只告警）
    if EXPECTED_RATE_HZ > 0 and dur_data > 0:
        expected_min = int(dur_data * EXPECTED_RATE_HZ *
DENSITY_FACTOR_MIN)
        if n_data < expected_min:
            print(f"[WARN] 样本密度偏低：实际 {n_data}，阈值 {expected_min}（期
望频率 {EXPECTED_RATE_HZ}Hz）")
            # 这里只做告警，不拉红
            # 如需严格，改成 all_pass = False

    print("[RESULT] PASS" if all_pass else "[RESULT] NOT PASS")
    sys.exit(0 if all_pass else 1)

if __name__ == "__main__":
    main()
```

ContentView.swift

```
import SwiftUI

#if targetEnvironment(simulator)
let defaultHost = "127.0.0.1"
#else
let defaultHost = "192.168.31.22"   // ← 改成 Mac 局域网 IP
#endif

fileprivate let portFormatter: NumberFormatter = {
    let f = NumberFormatter()
    f.numberStyle = .none
    f.minimum = 1
    f.maximum = 65535
    return f
}()

struct ContentView: View {
    @AppStorage("udpHost") private var udpHost: String = defaultHost
    @AppStorage("udpPort") private var udpPort: Int = 9001

    @State private var sender = UdpSender(host: defaultHost, port: 9001)
    @State private var timer: Timer?
    @State private var isSending = false
    @State private var sentCount: Int = 0
    @State private var lastSent: Date?

    var body: some View {
```

```
28          NavigationView {
29              Form {
30                  // 目标配置
31                  Section("UDP 目标") {
32                      TextField("Host (IPv4 或 主机名)", text: $udpHost)
33                          .textContentType(.URL)
34                          .keyboardType(.numbersAndPunctuation)
35
36                      TextField("Port", value: $udpPort, formatter:
    portFormatter)
37                          .keyboardType(.numberPad)
38
39                      Button("应用设置") {
40                          let p = UInt16(clamping: udpPort)
41                          print("APPLY host=\(udpHost) port=\(p)")
42                          sender.update(host: udpHost, port: p)
43                      }
44                  }
45
46                  // 连续发送控制：单独一行一个按钮
47                  Section("连续发送") {
48                      if isSending {
49                          Button("停止") {
50                              print("STOP button tapped")
51                              stopSending()
52                          }
53                          .buttonStyle(.borderedProminent)
54                          .tint(.red)
55                      } else {
56                          Button("开始每秒发送") {
57                              print("START button tapped")
58                              startSending()
59                          }
60                          .buttonStyle(.borderedProminent)
61                      }
62                  }
63
64                  // 发送一次：单独一行
65                  Section("调试/单次发送") {
66                      Button("发送一次") {
67                          sendOnce()
68                      }
69                      .buttonStyle(.bordered)
70                  }
71
72                  // 状态
```

```
                Section("状态") {
                    HStack {
                        Circle()
                            .fill(isSending ? .green : .gray)
                            .frame(width: 10, height: 10)
                        Text(isSending ? "正在连续发送" : "已停止")
                    }
                    Text("当前目标: \(udpHost):\(udpPort)")
                    Text("累计发送: \(sentCount) 条")
                    Text("最近一次: \(lastSent?.formatted(date: .omitted, time: .standard) ?? "—")")
                        .foregroundStyle(.secondary)
                }
                // ===== 新增: 实验标注 =====
                Section("实验标注") {
                    Button("基线开始") {
                        sendMarker("baseline_start")
                    }
                    Button("刺激开始") {
                        sendMarker("stim_start")
                    }
                    Button("刺激结束") {
                        sendMarker("stim_end")
                    }
                    Button("干预开始") {
                        sendMarker("intervention_start")
                    }
                    Button("干预结束") {
                        sendMarker("intervention_end")
                    }
                }
            }
            .navigationTitle("PolarBridge")
        }
    }

    // 发送一次（独立函数，便于复用）
    private func sendOnce() {
        let now = Date()
        let ts = now.timeIntervalSince1970
        let msg = #"{"type":"heartbeat","t":\#(ts)}"#
        print("SEND-ONCE \(msg)")
        sender.send(msg)
        sentCount += 1
        lastSent = now
    }
```

```
118
119     private func startSending() {
120         stopSending() // 防重入
121         isSending = true
122
123         let t = Timer.scheduledTimer(withTimeInterval: 1.0, repeats: true)
    { _ in
124             let now = Date()
125             let ts = now.timeIntervalSince1970
126             let msg = #"{"type":"heartbeat","t":\#(ts)}"#
127             print("TICK \(msg)")
128             sender.send(msg)
129             sentCount += 1
130             lastSent = now
131         }
132         RunLoop.main.add(t, forMode: .common) // 避免交互时暂停
133         self.timer = t
134         print("TIMER scheduled -> \
    (Unmanaged.passUnretained(t).toOpaque())")
135     }
136
137     private func stopSending() {
138         isSending = false
139         if let t = timer {
140             print("TIMER invalidate -> \
    (Unmanaged.passUnretained(t).toOpaque())")
141             t.invalidate()
142         }
143         timer = nil
144     }
145     // 在 ContentView 里加上这个辅助函数
146     private func sendMarker(_ label: String) {
147         let obj = #"{"type":"marker","label":"\#(label)"}"#
148         print("MARKER \(label)")
149         sender.send(obj)
150     }
151 }
```

UdpSender.swift

```
1  import Foundation
2  import Network
3
4  final class UdpSender {
5
```

```swift
    private var connection: NWConnection
    private var host: NWEndpoint.Host
    private var port: NWEndpoint.Port

    init(host: String, port: UInt16) {
        self.host = NWEndpoint.Host(host)
        self.port = NWEndpoint.Port(rawValue: port)!
        let params = NWParameters.udp
        self.connection = NWConnection(host: self.host, port: self.port, using: params)
        self.connection.start(queue: .global(qos: .utility))
    }

    /// 运行时更新目标地址（若未变化则不重建连接）
    func update(host: String, port: UInt16) {
        let newHost = NWEndpoint.Host(host)
        guard let newPort = NWEndpoint.Port(rawValue: port) else { return }
        if newHost == self.host && newPort == self.port { return }
        // 先取消旧连接
        self.connection.cancel()
        // 建立新连接
        self.host = newHost
        self.port = newPort
        let params = NWParameters.udp
        self.connection = NWConnection(host: self.host, port: self.port, using: params)
        self.connection.start(queue: .global(qos: .utility))
    }

    func send(_ text: String) {
        guard let data = text.data(using: .utf8) else { return }
        connection.send(content: data, completion: .contentProcessed { _ in })
    }
}
```

附2： 与后续阶段的衔接

- **阶段 4｜接入 Verity（PPI/HR）**
  - iOS 端以 Polar SDK 采集 PPI、HR，序列化为 JSON：
    `{"version":1,"stream":"ppi","packet_id":...,"payload":{"rr_ms":...,"hr_bpm":...}}`

    - 沿用当前桥接与 SOP，不变更 LSL 层与录制层。
- **阶段 6｜接入 H10（ECG）**
    - iOS 端或 macOS 端采集 ECG（按你的设备与 SDK 选择），以 130–1000 Hz 推送
    - 评估 UDP 是否满足带宽与可靠性；如需更稳，可在桥接器旁再加一条基于 TCP/WebSocket 的通道，仅用于高频原始波形。
    - 标记流复用当前 `PB_MARKERS`，保证实验段落可对齐。