“**Build with Us | Deep Dive: Transforming your Data with Code Repositories**”是 learn.palantir.com 平台上的一门深度实战课程，由 **Ontologize** 团队提供教学指导 1。该课程专注于教授用户如何使用 Palantir Foundry 中的 **Code Repositories**（代码存储库 ） 应用，通过编写代码的方式构建生产级的数据转换管道 1。
以下是根据来源对该课程内容的详细解释：

1. 核心定位：代码驱动的数据处理 (Pro-code)

- 工具对比：Foundry 提供两种构建数据管道的方式：**Pipeline Builder**（适合非代码开发者）和 **Code Repositories**（适合偏好代码的开发者）1。
- 开发环境：Code Repositories 是一个基于 Web 的集成开发环境 **(IDE)**，具备 Git 集成、计算配置文件（Compute Profiles）访问等常见 IDE 功能 1, 2。
- 技术栈：课程主要使用 **PySpark** 结合 Foundry 的 API 来编写转换逻辑 2, 3。

2. 业务场景：保险理赔分析

- 背景：用户扮演全球保险公司的理赔处理人员，任务是向首席财务官（CFO）汇报各业务线的年度业绩 4。
- 原始数据：处理两个核心数据集：历史理赔数据 **(Claims)** 和相关的保单数据 **(Policies)** 4, 5。

3. “深度潜入”涵盖的技术环节

该课程通过一个完整的开发周期，带用户练习最常见的开发模式：

- 环境初始化：
- 创建专门的项目和文件夹结构（如 logic、data/raw、data/prepared）来组织资产 4, 6。
- 初始化 Python 转换存储库，并学习如何通过资源 **ID (RID)** 或文件系统路径引用输入数据集 6-8。
- 核心转换操作：
- 数据清洗 **(Cleaning)**：使用正则表达式清除字符串中的杂质（如 # 符号），并将日期字段从字符串类型转换 **(Cast)** 为标准日期格式 9-11。
- 数据过滤 **(Filtering)**：利用 PySpark 函数根据布尔值列（如 is_accepted）筛选有效行 12, 13。
- 多表关联 **(Joining)**：通过 transform_df 装饰器引入多个输入，利用共享键（如 policy_id）执行左连接 **(Left Join)**，从而将保单中的“业务线 (LOB)”信息整合到理赔数据中 14-16。
- 数据聚合 **(Aggregation)**：使用 groupBy 方法计算各业务线的平均理赔成本 17, 18。
- 代码发布与构建：
- 预览 **(Preview)**：在不写入磁盘的情况下查看转换后的数据样貌 9, 19。
- 提交 **(Commit)**：保存代码状态并触发持续集成 **(CI)** 检查，验证存储库的一致性 19, 20。
- 构建 **(Build)**：启动 Spark 作业执行逻辑，并在 Foundry 文件系统中实例化输出数据集 19, 21, 22。

4. 协作与分支管理 (Collaboration)

作为深度教程，它还涵盖了工业级开发的最佳实践：

- 分支保护 **(Branch Protection)**：将 master 分支设为只读，以保护生产级代码 23, 24。
- 分支开发：从主分支切出新分支进行功能开发 18。

- 合并流程 **(Merge/Pull Request)**：通过合并请求 **(Pull Request)** 描述变更，并利用“压缩并合并 (Squash and merge)”等 Git 技术将代码合并回主干 25-27。

5. 最终产物与可追溯性

- 数据血缘 **(Data Lineage)**：在数据血缘应用中观察从原始 CSV 到聚合报告的全路径 22, 28。
- 伪代码查看：对于由代码生成的资产，用户可以在 Data Lineage 中直接查看其背后的 PySpark 逻辑 28。

总结来说，这个标题代表了一次从零到一的 **Pro-code** 开发实践。它不仅教导如何编写 PySpark 代码，更重要的是教导如何在 Foundry 的受控环境下管理代码生命周期、数据质量和团队协作。