

中国科学院大学

Object-Oriented Programming 报告

项目名称	Qlib		
地址	https://github.com/microsoft/qlib		
姓名	张超超	学号	2023K8009906024
邮箱	zhangchaochao23@mails.ucas.ac.cn		

笔者于 2025-2026 秋季学期学习中国科学院大学王伟老师的面向对象程序设计课程，并将阅读 Qlib 项目源码的心得体会记录在本报告中。读者对我在报告中的思考有任何问题都欢迎与我交流。

目录

1	Qlib 简介	3
1.1	framework	3
1.2	角色分析	4
2	需求分析与建模	4
2.1	功能性需求	5
2.1.1	端到端研究流水线支持	5
2.1.2	可配置、可替换的数据接入层	5
2.1.3	表达式驱动的特征计算	5
2.1.4	最小必要的的数据质量保证	6
2.1.5	实验管理与可复现	6
2.2	非功能性需求分析	7
2.3	面向对象建模	8
2.4	设计决策与权衡（为什么这样建模）	9
2.4.1	复杂业务清洗最小化并留给用户自定义	9
2.4.2	表达式驱动与惰性求值以在性能与灵活性之间取得平衡	10
2.4.3	模块化抽象接口以支持可复现研究与组件可替换性	10
3	核心流程设计分析	10
3.1	数据处理与特征工程流程	11
3.2	数据集封装与训练/验证/测试划分	12
3.3	模型训练与预测执行流程	12
3.4	策略生成与回测执行	12
3.5	结果记录与实验复现	13
4	高级设计意图分析	13
4.1	核心思想：表达式驱动的数据计算	13
4.2	策略模式 + 插件式扩展	13
4.3	高内聚，低耦合	13

5	扩展点需求分析与建模	14
5.1	代码仓库	14
5.2	建模分析	15
6	结语	16

1 Qlib 简介

Qlib 是一个由 Microsoft Research 团队开发的开源、面向人工智能的量化投资平台，旨在通过人工智能技术释放潜能、赋能研究，并在量化投资中创造价值，覆盖从策略构思到实盘落地的完整过程。

其支持多种机器学习建模范式，包括监督学习、市场动态建模以及强化学习。

1.1 framework

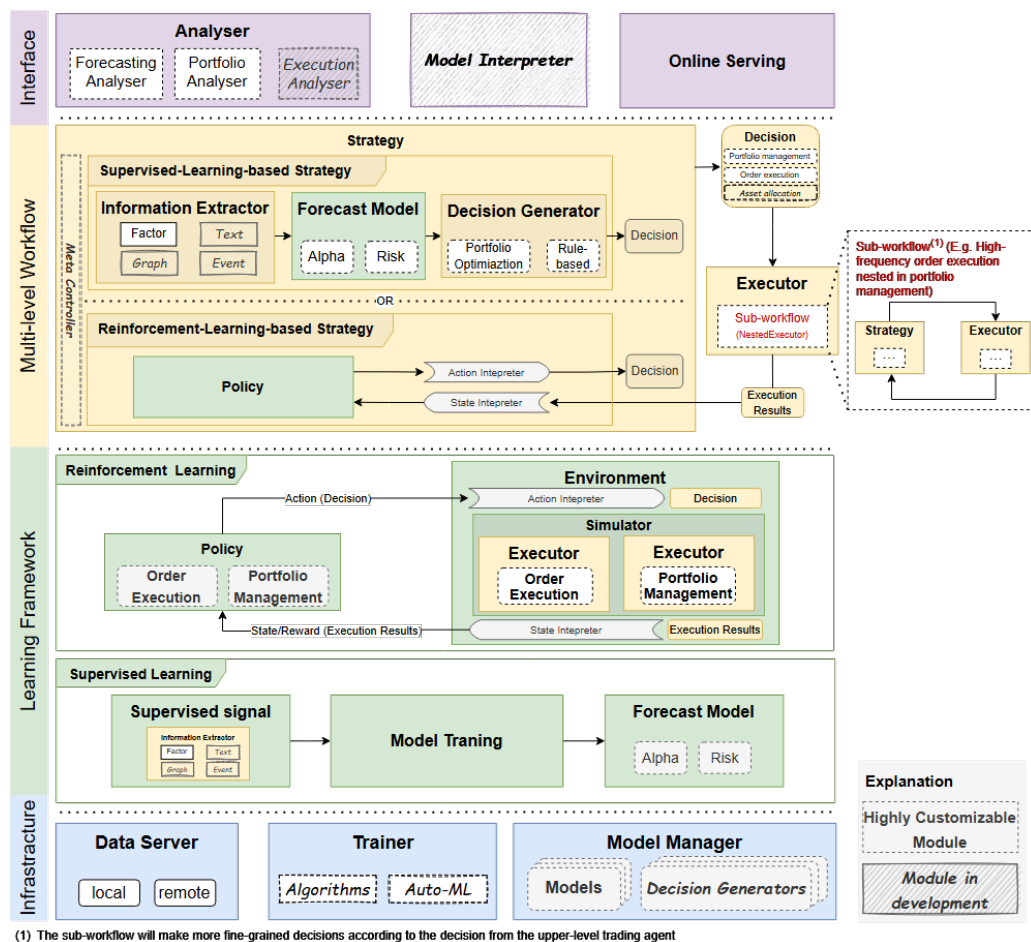


图 1: framework

- 接口层旨在为底层系统提供友好的用户界面。

Analyser 模块将为用户提供关于预测信号、投资组合以及执行结果的详细分析报告

- 工作流层覆盖了量化投资的整个流程。

信息提取器负责为模型提取所需数据。

- 学习框架层预测模型和交易代理是可训练的。

- 基础设施层为量化研究提供底层支持。

DataServer 为用户管理与获取原始数据提供高性能的基础设施。

Trainer 提供灵活的接口，用于控制模型的训练过程，从而支持算法对训练流程的调控。

1.2 角色分析

Qlib 系统包含多个参与角色，各自的职责定位如下：

- **Researcher / Quant (研究员 / 量化研究员)**

负责因子研究、特征工程、模型训练、模型评估以及投资组合生成，是系统的主要使用者。

- **Data Engineer (数据工程师)**

负责数据的获取、清洗及点位时校正，保证研究过程不出现未来信息泄漏。

- **Operator (运维人员)**

负责系统插件（模型 / 策略 / 执行器等）的注册与扩展，保证系统环境可复用、可拓展。

- **Trader / Execution System (交易执行系统)**

在模型部署和实时推断后，接收模型信号并执行真实或模拟交易。

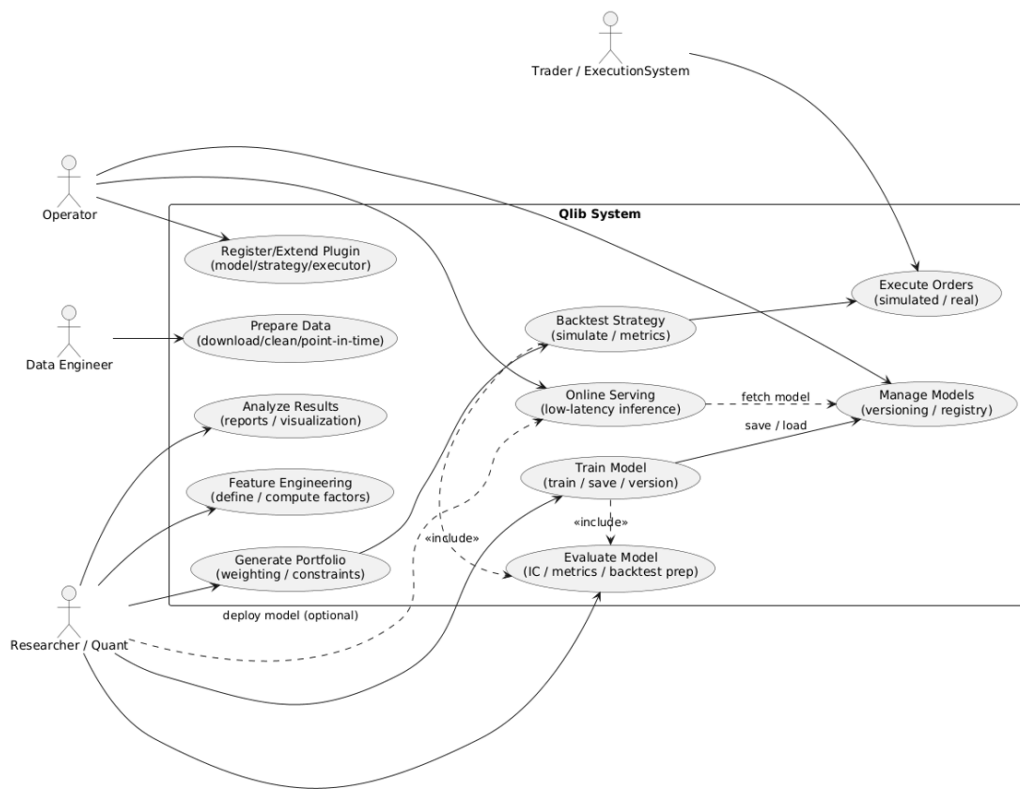


图 2: Qlib 用例图

2 需求分析与建模

Qlib 以量化研究闭环为核心，从数据 → 特征 → 模型 → 回测 → 交易，整个系统围绕“面向对象层次化抽象”设计，

建模核心思想是：

用一组“稳定的抽象父类”定义研究流程，再通过“可插拔子类”完成算法/清洗策略/模型策略的扩展。

2.1 功能性需求

2.1.1 端到端研究流水线支持

Qlib 提供了从数据准备到模型回测的一体化端到端研究流水线，使研究者无需编写大量“glue code”即可完成量化研究全过程。

具体而言，该流水线覆盖了原始市场数据读取、特征表达式计算、训练/验证/测试数据集构建、模型训练与预测、结果记录和回测评估等关键阶段，并将其封装为统一的可复用接口。

首先，系统通过 `qlib/scripts/get_data.py` 以及

`qlib/cli/data` 模块实现原始行情数据的下载与标准化存储，其中包括对点位时（Point-in-Time）数据一致性的支持，避免未来数据泄漏。

在数据集构建阶段，Qlib 定义了可复用的数据集抽象，并在时间维度上自动划分训练集、验证集和测试集。直接运行 `qrun`（即 `qlib/cli/run.py`）加载 YAML 流程配置，即可自动完成模型训练、保存与预测，无需研究者在脚本中手动串联各阶段。

训练完成后，系统会将预测结果记录至工作流目录，并调用回测与评估模块（如 `qlib/backtest/backtest.py`）实现策略回测、收益风险指标计算及可视化分析。至此，研究者即可基于统一的实验配置文件快速实现模型迭代与策略评估。

2.1.2 可配置、可替换的数据接入层

Qlib 在数据接入部分提供了高度可配置与可替换的抽象层，以屏蔽底层数据源的格式差异，实现统一的数据读取接口。

研究者可以将来自 CSV 文件、二进制数据存储或数据库等多种来源的市场数据，统一转换为 Qlib 的标准数据格式，而无需修改后续特征构建与模型训练逻辑。

具体而言，Qlib 在 `qlib/data/dataset/loader.py` 中定义了 `DataLoader` 抽象接口，用于统一描述数据加载行为：

```
1 class DataLoader(abc.ABC):
2     """
3     DataLoader is designed for loading raw data from original data source.
4     """
5
6     @abc.abstractmethod
7     def load(self, instruments, start_time=None, end_time=None) -> pd.DataFrame:
```

用于从框架准备好的本地数据集结构中按需加载时间序列特征与目标值。同时，用户可以通过继承 `DataLoader` 接口，自定义数据读取方式，例如从专有数据库、在线行情源或异构文件系统中取数，并无缝接入既有研究流水线。

2.1.3 表达式驱动的特征计算

Qlib 在特征工程阶段引入了基于表达式（Expression）与领域特定语言（DSL）的特征计算机制，研究者可以使用类似于因子语言的形式对特征进行描述，例如 `EMA($close, 12)`、`Ref($close, -1)` 等。

这类表达式能够抽象并复用特征构造逻辑，而无需直接手写面向数据表格或数组操作的 Python 代码，从而显著提高因子设计与快速迭代的效率。

当模型或数据集实际调用特征时，系统再触发求值过程，从而减少不必要的中间数据物化与磁盘 I/O。

研究者可以通过调用 `Dataset` 对象的 `features()` 接口按需加载特征，其中 `disk_cache` 参数用于控制特征缓存策略：若开启磁盘缓存，特征值将在首次计算后写入本地存储，加速后续访问；若关闭，则保持纯内存或即时计算模式，适用于快速测试与小规模实验。

在具体数据处理流程中，表达式解析与求值机制与数据集加载模块紧密结合，确保特征计算严格遵循时间一致性与点位时（Point-in-Time）原则，从而避免未来数据泄漏问题。同时，表达式驱动的特征计算天然具备可组合性，研究者可通过嵌套和组合表达式，高效构造多尺度、多维度的复合因子。

2.1.4 最小必要的数据质量保证

Qlib 在数据质量控制方面遵循“最小必要处理”原则，即框架只负责确保训练与回测管线在基本数据一致性条件下能够稳定运行，而不对数据进行业务逻辑层面的深度清洗与重构。框架重点解决的是时间索引对齐、基础缺失值处理、字段表达式计算一致性保证等低层数据可靠性问题，从而避免模型训练过程中因 NaN、索引错位或数据无法对齐而导致的流程中断。

具体而言，Qlib 在 `qlib/data/dataset/processor.py` 中定义了多级数据处理抽象。其中 `processor` 与 `qlib.data.inst_processor` 作为可插拔式的数据处理插件，被应用于特定字段或特定股票（instrument）的数据清洗过程。

此外，`qlib/data/dataset/loader.py` 和 `qlib/data/dataset/handler.py` 在加载数据与构建样本时会自动执行日历对齐（calendar alignment），确保不同股票横截面在同一交易日具有一致的时间索引。

需要强调的是，Qlib 并不尝试解决会计科目映射、财报补全、分红送股调整等复杂的业务级数据治理问题，这些部分通常由数据供应商或研究者自身根据研究目标完成。

框架仅提供标准化的处理扩展点，使用户能够在现有流水线上以插件形式挂载自定义数据处理逻辑，而无需修改核心框架代码。

因此，Qlib 的数据质量保证策略在保证训练与预测管线鲁棒性的前提下，保持了数据处理模块的可解释性与可扩展性，避免过度“黑箱式”清洗带来的研究偏差，并允许研究者根据不同数据源与策略特征灵活定制更高层次的数据处理方案。

2.1.5 实验管理与可复现

量化研究中，同一策略在不同数据版本、训练超参、特征定义乃至回测配置下往往会产生显著差异。因此，系统必须提供可追踪与可复现的实验管理能力，以避免“模型行为无法重现”的实验黑箱现象。

Qlib 提供了完整的 Experiment 管理框架，通过 Recorder 组件实现实验的配置保存、模型存档、指标记录与结果回溯等功能：

- **配置记录（Config Tracking）**：自动持久化本次实验涉及的数据描述、特征工程配置、模型结构与训练超参数。
- **模型与权重存储（Model Checkpointing）**：在不同训练阶段保存模型参数，以便后续直接加载进行复盘与再训练。
- **结果记录与回溯（Result Recording & Playback）**：包括预测结果、回测收益表现（如 IC/IR、收益曲线、行业中性 IC 等），支持实验间横向对比。
- **实验目录结构化管理（Structured Experiment Workspace）**：提供统一文件组织规范，避免本地实验输出凌乱难以检索。

典型使用方式为通过 R（Recorder）对象记录训练与回测过程，例如：

```
1 with R.start(experiment_name="exp_dnn_factor") as recorder:
2     model.fit(dataset)
3     recorder.save_objects(model=model, dataset=dataset)
4     recorder.log_metrics(**evaluate(model, dataset))
```

通过上述机制，研究者可以：

1. 回溯任意一次实验的环境、模型、数据与表现；
2. 在多实验之间进行客观可重复的性能对照；
3. 构建与团队成员共享的“可复现实验谱系”。

这使得 Qlib 不仅是一个策略开发/回测工具，更是支持完整研究闭环的可复现性平台。

2.2 非功能性需求分析

在 Qlib 的整体设计中，其核心非功能性目标可总结为：在可接受的资源消耗下支持大规模量化研究任务；通过模块化与插件化机制保证系统可扩展与可重用性；通过统一的实验记录机制确保训练与回测的可复现性；并为研究人员提供低工程负担的特征构建与模型实验环境。

这些目标贯穿于数据层、特征层、模型层与实验管理层各部分的接口抽象与代码组织中。

(1) 性能与可扩展性 量化因子研究典型地面对 $T \times S \times F$ （时间、标的、特征）高维数据，其中 T 可覆盖十年以上交易日、 S 包含数千股票或期货合约、 F 涉及百级以上特征变量。Qlib 在数据加载与特征计算层采用“基于需求的按块访问”与“缓存驱动的增量计算”来保证高性能。

首先，原始数据读取在 `qlib/data/data/handler.py` 与 `qlib/data/dataset/loader.py` 中实现了按日历与标的的分块的增量访问，避免数据全量加载至内存。

其次，特征表达式（如 EMA, Ref）的求值过程由 `FeatureProcessor`（文件：`qlib/data/dataset/processor.py`）驱动，并通过 `disk_cache`（缓存目录在 `./qlib/cache`）实现持久化缓存，用于避免重复计算。

整体而言，这种“动态加载 + 增量计算 + 缓存复用 + 并行执行”的策略使系统能够在不显著增加复杂性的情况下支持数百 GB 级别数据规模。

(2) 模块可复用性与可扩展性 Qlib 的架构采用典型的“接口-实现-插件替换”模型，使用户可以像拼装积木一样替换任意研究环节。例如：

- **数据接入层**遵循 `DataLoader` 接口（`qlib/data/dataset/loader.py`），用户可通过继承此接口接入数据库、CSV、本地缓存或第三方行情源；
- **特征处理与清洗**由 `Processor` 管线抽象（`qlib/data/dataset/processor.py`），允许每个 instrument 配置独立清洗策略；
- **模型层**遵循统一的 `Model` 接口（`qlib/model/base.py`）。
- **策略与执行层**定义在 `qlib/strategy` 用于从因子预测信号到最终持仓构造的分层抽象。

这种分层接口化设计使研究者能够仅替换所需模块，而无需重新实现整条研究与回测流程，大大提升了模型创新与因子组合研究的效率。

(3) 实验可复现性与可追踪性 量化研究的核心要求是可复现性与可审计性。Qlib 提供了统一的实验管理框架，核心组件为 `Recorder` 与 `Experiment`（文件位置：`qlib/workflow/recorder.py` 与 `qlib/workflow/expm.py`）。它将以下研究产物统一纳入记录范围：

- 数据配置与特征表达式；
- 模型训练超参数与权重快照；
- 预测输出（即“信号”）；
- 回测收益、风险、指标曲线；

- 训练日志与环境信息。

每次实验均可通过唯一的 `experiment_id` 重现完整研究过程，实现实验对比、版本回退与历史审计。

(4) **研究便捷性** Qlib 面向量化研究人员的一个重要设计目标是降低工程复杂度。其 **表达式驱动** 的特征计算 (`qlib/data/dataset/handler.py`) 允许研究者使用 因子 DSL (如 `MA(close, 20)` 或 `RSI(close, 14)`) 直接声明因子逻辑，而无需手动进行字段计算与存储管理。

结合统一的 **Dataset** 管线与可替换模型接口，研究者能够以最少的样板代码快速构建不同因子组合与预测模型，形成以算法创新为主导、以工程机制为支撑的高效研究方式。

2.3 面向对象建模

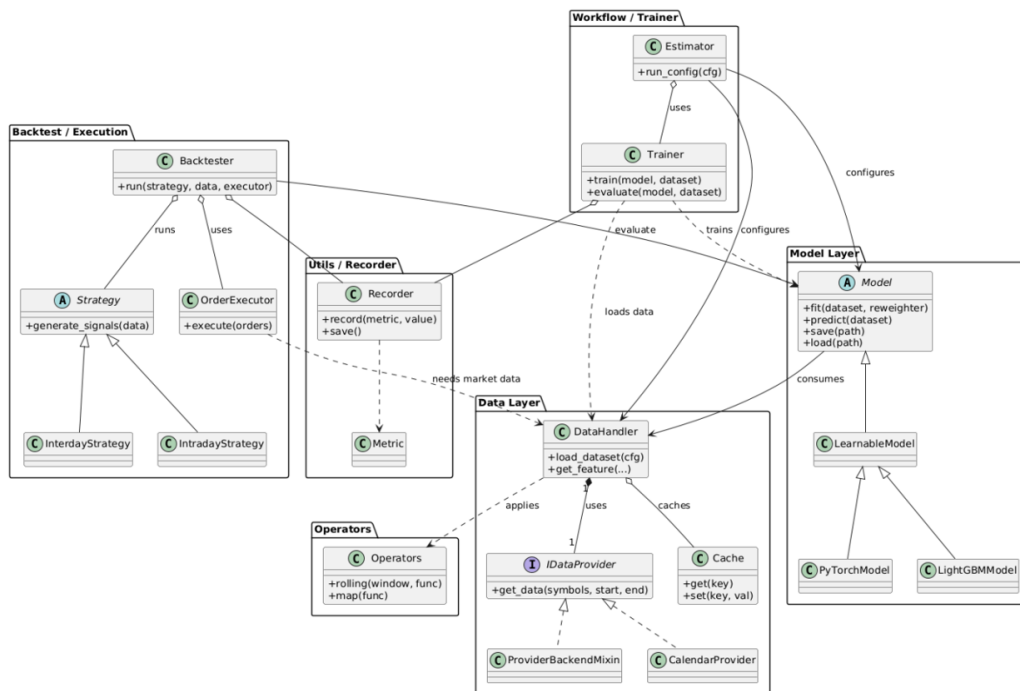


图 3: UML

如图展示了 Qlib 系统在研究—训练—回测全过程中的核心类之间的调用关系与数据流向。

从整体架构来看，Qlib 将量化研究与交易流程划分为数据层 (Data Layer)、模型层 (Model Layer)、 workflow 与训练层 (Workflow / Trainer)、回测与执行层 (Backtest / Execution) 以及实验记录层 (Utils / Recorder) 五个相互解耦但可协同工作的功能域。这种分层式架构使框架具备良好的可替换性与可扩展性，同时确保研究过程能够被完整复现。

数据层 (Data Layer) 数据层是系统运行的基础，其核心类为 `DataHandler`，对应源码位置 `qlib/data/dataset/handler.py`。该组件负责根据配置加载数据集、对齐交易日历并提供特征字段访问接口。

抽象，支持从本地缓存、GBDT 生成的二进制数据、数据库等多种数据源获取行情与基本面信息。

为降低 I/O 成本，数据访问会经过 `Cache` (`qlib/data/cache.py`)，实现特征级或数据块级的本地缓存机制。

模型层 (Model Layer) 模型层由抽象类 `Model` (`qlib/model/base.py`) 定义统一训练与预测接口，包括 `fit`、`predict`、`save`、`load` 等方法。

模型层通过标准化的数据与预测接口，实现模型间的可替换性，使研究者能够在不改变数据流与回测逻辑的情况下切换模型结构。

工作流与训练层(Workflow / Trainer) 工作流层对模型训练过程进行封装,核心类为 `Trainer` 与 `Estimator`。

在一项实验中, `Estimator` 根据配置选择具体模型,并调用 `Trainer` 使用 `DataHandler` 中构造的数据集进行模型训练与评估。此层负责训练阶段的组织与调度,而不关心具体模型的内部细节,从而维持架构的低耦合性。

回测与执行层 (Backtest / Execution) 回测阶段采用 `Backtester` (`qlib/backtest/backtest.py`) 运行策略逻辑。

策略由抽象类 `Strategy`(`qlib/strategy/base.py`)定义,子类如 `InterdayStrategy` 与 `IntradayStrategy` 根据模型预测信号生成交易指令,随后由 `OrderExecutor` 负责将指令映射为持仓变化并计算交易成本、滑点等指标。

该体系使策略逻辑、信号预测与交易执行过程互相独立、可独立替换。

实验记录层 (Utils / Recorder) 实验记录能力由 `Recorder` (`qlib/workflow/recorder.py`) 提供,用于保留训练日志、模型权重、预测信号、回测结果与评价指标 (`Metric`)。

所有过程可通过唯一实验编号回放,从而实现研究成果的可复现与审计。

通过上述分层设计, `Qlib` 将数据处理、模型训练、策略生成、回测执行与实验记录有机串联,并通过统一接口实现模块之间的低耦合与高可替换性。这使研究者能够在不改变整体流程的前提下灵活尝试不同数据源、不同模型与不同策略逻辑,从而加速量化因子与模型在研究—验证—落地过程中的迭代效率。

2.4 设计决策与权衡 (为什么这样建模)

`Qlib` 的架构体现了“通用基础设施 + 可自由组合扩展点”的核心设计理念,即框架不试图为研究者定义统一的量化研究范式,而是通过稳定接口、惰性计算与可追踪实验体系,提供可被长期复用的基础研究环境。该设计在数据层、特征表达层与模型及实验管理层分别体现出不同的取舍。

2.4.1 复杂业务清洗最小化并留给用户自定义

金融数据处理具有显著的业务异质性:不同市场的停复牌规则、不同供应商的复权口径、不同研究团队对财务指标修正的理解均可能存在差异。而一旦框架在数据准备阶段嵌入过多“默认业务逻辑”,不仅会限制适配范围,还可能直接影响因子研究结果的可控性。因此, `Qlib` 在数据层采用“最小必要清洗 (Minimal QA)”策略。

框架仅保证数据在时间维度上可对齐、在字段访问上可表达、在基本缺失值上可处理。例如:

- 字段与特征处理通过 `Dataset` 中的 `processor` 链 (`qlib/data/dataset/processor.py`) 完成,用户可对不同标的设定独立处理逻辑。
- 更复杂的去极值、财务指标修复、行业分类处理等不被框架自动执行,而是通过 `processor` 或外部数据源由研究者自行定义。

这一策略的权衡在于:框架不保证“业务含义的正确性”,但保证“数据结构的可研究性”。这使 `Qlib` 能够适配不同市场场景 (A 股 / 港股 / 美股 / 加密货币),并支持不同风格的因子工程。

2.4.2 表达式驱动与惰性求值以在性能与灵活性之间取得平衡

量化研究中常见的数据规模为 $T \times S \times F$ ，在多资产及高频场景下，特征存储可以轻松达到 TB 级。

若框架强制进行“离线因子构建并整体缓存”，不仅存储成本高昂，也会使研究迭代效率显著下降。Qlib 因此选择了“表达式 DSL + 惰性求值 (Lazy Evaluation) + 可控缓存”策略。

特征表达式由 `qlib/data/ops.py` 中的算子系统定义，并在 `qlib/data/data.py::D.features()` 中执行实际求值。

求值过程遵循：

- 当特征未被访问时，不进行计算；
- 当同一表达式在不同实验中重复使用时，通过 `disk_cache` 与 `MemoryCache` 减少重复计算；
- 当研究者调整因子结构时，无需重新构建完整特征库，只需重新解析依赖链。

这种“只算需要的 + 能缓存的 + 不强制离线化”的机制使得研究者可以：

- 快速试验新因子形式，而不需要全量重新生成；
- 在特征数量快速扩增时保持可控存储开销；
- 根据硬件环境调节“计算优先”或“存储优先”的策略。

其设计哲学类似于“表达式即计算图，数据加载即执行调度”，在灵活性与性能之间取得可配置的中间解。

2.4.3 模块化抽象接口以支持可复现研究与组件可替换性

不同研究团队的方法差异主要体现在模型、特征构造、交易逻辑和执行策略上，因此框架若要支持长期研究演进，就必须确保组件可替换而不破坏整体结构。Qlib 在体系架构中通过抽象接口层实现这一点：

- 数据与特征： `DataLoader` 与 `Dataset`
- 模型：统一继承自 `Model`（见 `qlib/model/base.py`），如 `LightGBMModel` 与 `TFTModel` 均只需实现 `fit/predict`
- 实验可复现： `Recorder/Experiment`（`qlib/workflow/record_temp.py` 提供实验生命周期管理）
- 回测执行与策略层： `Strategy + OrderExecutor + Backtester`

由于接口清晰：

- 研究者可在不修改数据代码的情况下替换模型；
- 可在不改变预测输出格式的情况下替换回测逻辑；
- 所有实验配置、模型权重、预测信号、回测结果均可被完整记录和复现。

这使研究过程具备科研所需的“可推导、可回放、可对照”的可追踪性特征。

3 核心流程设计分析

Qlib 的研究流程围绕从原始行情数据出发，构造特征、切分数据集、训练预测模型并进行策略回测的完整闭环展开。整体数据流可描述为：

Raw Data → Provider → DataHandler → Dataset → Model → Strategy → Backtest → Recorder/Report

在实际实现中，各阶段组件之间通过清晰的接口协议连接，既避免耦合又便于替换与扩展。

3.1 数据处理与特征工程流程

数据层的核心是 `DataHandler`，其负责对原始行情数据进行统一格式化、特征表达式求值与时间对齐。源码位于：

- `qlib/data/dataset/handler.py`
- `qlib/data/dataset/processor.py`
- `qlib/data/ops.py`

核心类定义如下：

```
1 // qlib/data/dataset/handler.py
2 class DataHandler(DataHandlerABC):
3     """
4     负责：
5     1) 加载基础行情数据（通过 IDataProvider -> ProviderBackendMixin）
6     2) 计算/加载特征（Expression/OPS 形式惰性求值）
7     3) 对齐并构建训练/验证/测试所需数据切片
8     """
9     def fetch(self, selector, level, col_set=CS.ALL, data_key=DK.L, **kwargs):
10         data_df = self._data_map[data_key]
11         data_df = fetch_df_by_col(data_df, col_set)
12         data_df = fetch_df_by_index(
13             data_df,
14             selector,
15             level=level,
16             fetch_orig=self.fetch_orig,
17         )
18
19         return data_df
20     def process(self, **kwargs):
21         _shared_df = self._run_proc_l(self._data, self.shared_processors)
22         _infer_df = self._run_proc_l(_shared_df, self.infer_processors)
23         _learn_df = self._run_proc_l(_infer_df, self.learn_processors)
24         self._infer = _infer_df
25         self._learn = _learn_df
26         return self
```

其内部调用链为：

1. **DataLoader** 从底层数据源加载输入数据，并将其统一组织为 `(datetime, instrument)` 的 `MultiIndex DataFrame`。
2. **DataHandler** 对数据进行列集管理(特征 / 标签 / 原始数据划分)并提供统一抽取接口 `fetch()`(`qlib/data/dataset/`).
3. **DataHandlerLP** 在此基础上引入三段式可学习处理流水线：
 - (a) `shared_processors`：对原始数据进行基础一致性处理（如缺失填补、交易日对齐）。
 - (b) `infer_processors`：生成推理用数据，确保不引入未来信息（防止泄漏）。
 - (c) `learn_processors`：基于训练集统计量进行归一化、异常值截断等变换。
4. 模型、回测与滑窗训练所使用的数据均通过 `fetch(selector, level, col_set, data_key)` 从上述版本中按需提取。

3.2 数据集封装与训练/验证/测试划分

在数据处理完成后，由 Dataset 负责组织训练、验证和测试分段。

```
1 // qlib/dataset/dataset.py
2 class Dataset:
3     def __init__(self, handler, segments):
4         self.handler = handler          # 绑定 DataHandler
5         self.segments = segments        # 训练/验证/测试区间 (dict)
6
7     def prepare(self, segment):
8         # 基于时间段取得 X,y
9         X, y = self.handler.fetch(segment)
10        return DataHandlerLP(X, y)      # 构造成模型可消费结构
```

数据组织结构为：

```
1 Dataset
2     train → (X_train, y_train)
3     valid → (X_valid, y_valid)
4     test  → (X_test , y_test)
```

通过 segments 参数，用户可通过 YAML/JSON 配置直接指定回测与训练的时间窗口，实现严格的时间分割训练，避免未来函数问题。

3.3 模型训练与预测执行流程

模型层以 Model 抽象类为核心，所有模型（树模型、线性模型、深度模型）均实现统一接口。源码位于 qlib/model/base.py。

```
1 // qlib/model/base.py
2 class Model:
3     def fit(self, dataset):
4         raise NotImplementedError
5     def predict(self, dataset):
6         raise NotImplementedError
```

由于输入为 Dataset，因此任何模型（如 MLP、GRU、Transformer、Autoformer）均可无缝替换，只需实现同样的接口结构。

3.4 策略生成与回测执行

模型预测的收益率信号需要转化为投资组合权重策略，并在资金约束与交易成本模型下进行回测。相关模块位于：

- 策略：qlib/strategy/
- 执行器：qlib/backtest/executor.py
- 组合持仓演进：qlib/backtest/backtest.py

示例策略：

```

1 // qlib/strategy/strategy.py
2 class TopkDropoutStrategy(Strategy):
3     def generate_signals(self, pred):
4         # 对预测值进行排序, 构建 top-k 持仓组合

```

回测执行流程:

```

1 // qlib/backtest/executor.py
2 class BacktestExecutor:
3     def run(self, strategy):
4         # 根据信号, 成本模型, 仓位约束, 执行日/调仓日
5         # 驱动组合权重逐日演进

```

3.5 结果记录与实验复现

所有实验（模型配置、预测结果、回测结果、曲线与指标）通过 `Recorder` 统一存储：

- 位置: `qlib/workflow/record_temp.py`

使得相同配置可完全复现模型行为，满足研究可追踪性要求。

4 高级设计意图分析

4.1 核心思想：表达式驱动的数据计算

```

1 // qlib/data/dataset/processor.py
2
3 $close_0 / $close_5 - 1    →    计算 5 日收益率
4 Ref($volume, 1)           →    前1日成交量

```

这些表达式不会立即计算（惰性）

只有在训练需要数据时才执行（降低存储与 I/O 成本）

4.2 策略模式 + 插件式扩展

模块	基类	扩展方式
DataHandler	DataHandlerLP	自定义数据处理规则
Model	Model	接入任意 ML / DL 模型
Strategy	Strategy	任意投资信号生成方案
Executor	Executor	接入外部交易执行系统

4.3 高内聚，低耦合

关键解耦机制

数据与模型解耦（Dataset 是中间层）

预测与交易策略解耦（Signal 独立于 Model）

模型训练与实验流程解耦（Workflow 独立于 Model/Data）

研究者可单独更换模型，而无需重建数据与策略流程。

5 扩展点需求分析与建模

模型训练需要固定形状 (stock \times time) 张量, 补齐方式代表交易假设, 不同研究者选择不同。Qlib 设计上追求中立和通用性, 不希望把单一的金融假设 (例如 “停牌日填前值” 或 “volume 填 0”) 强加给所有用户或市场, 因此只提供低层 Fillna / Processor 基础, 而不内置强策略。

```
def data_merge_calendar(self, df: pd.DataFrame, calendars_list: List[pd.Timestamp]) -> pd.DataFrame:
    # calendars
    calendars_df = pd.DataFrame(data=calendars_list, columns=[self.date_field_name])
    calendars_df[self.date_field_name] = calendars_df[self.date_field_name].astype("datetime64[ns]")
    cal_df = calendars_df[
        (calendars_df[self.date_field_name] >= df[self.date_field_name].min())
        & (calendars_df[self.date_field_name] <= df[self.date_field_name].max())
    ]
    # align index
    cal_df.set_index(self.date_field_name, inplace=True)
    df.set_index(self.date_field_name, inplace=True)
    r_df = df.reindex(cal_df.index)
    return r_df
```

图 4: Caption

因此用户常常需要在 Qlib 之外自行清洗数据 (获取交易日历 \rightarrow reindex/对齐每只股票 \rightarrow 选择 OHLC 填充策略 (ffill, interpolate, shift) \rightarrow volume 填 0/NaN/均值/ffill \rightarrow 异常检测/剔除 \rightarrow 缺失率检查 \rightarrow 导入 Qlib。重复、易错、不可复用)。

故而笔者的希望基于此, 实现拓展市场行情数据质量治理增强模块 Market Data Quality Enhancer(MDWE):

实现交易日对齐 + 多策略填补 + 异常检测/修正 + 缺失率检测 (标记/删除) + 数据质量评分/报告 + 增量更新连成一套数据治理 (Data QA) 工具链, 显著提高研究数据的可信度与可复现性。

该拓展保持原则:

不改变 Qlib 默认行为 (默认数据加载/处理不调用本模块), 插件式 / 策略可配置, 每种填补/异常/评分/增量策略是独立类, 用户按需组合, 兼容多市场, 可审计 & 可复现, 增量安全。

5.1 代码仓库

```
1      qlib/
2      data/
3          processing_ext/          # 新建 MEQE
4              __init__.py          # 统一暴露的清洗接口列表
5              base.py              # 策略接口 & 清洗器抽象基类
6              align_calendar.py    # 对齐交易日历
7              ohlc_strategies.py   # OHLC (开高低收) 字段修复策略
8              volume_strategies.py # 成交量缺失处理
9              anomaly.py           # 异常点检测与处理
10             quality.py            # 数据质量评估
11             cleaner.py            # 整体清洗流程控制器
```

具体流程为:

- 1) 对齐交易日 \rightarrow AlignCalendar
- 2) 应用 OHLC 策略
- 3) 应用 Volume 策略
- 4) 异常检测 (可选)
- 5) 数据质量评分
- 6) 若缺失率超过阈值 \rightarrow 丢弃该股票

详情参见代码仓库。

5.2 建模分析

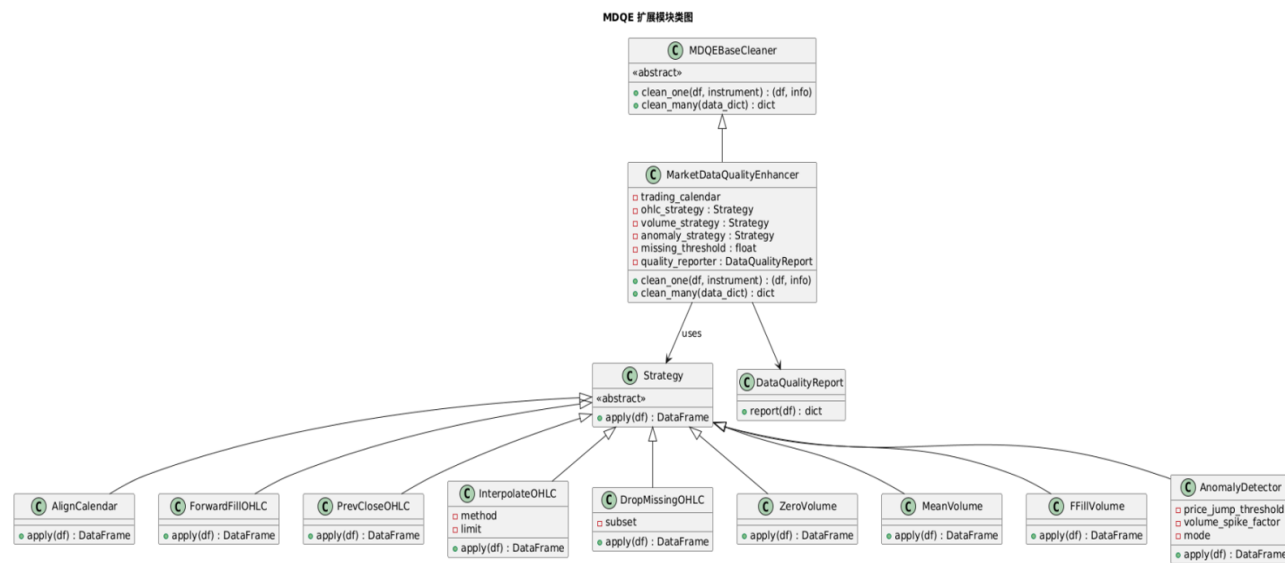


图 5: Caption

本模块采用可组合的策略（Strategy）模式实现市场数据质量增强（MDQE），其结构如下：

- **清洗器基类：** MDQEBaseCleaner 定义统一清洗接口，包括 clean_one 与 clean_many 方法，为具体清洗管线提供抽象框架。
- **清洗管线实现：** MarketDataQualityEnhancer 负责组织和调度各类清洗策略，内部包含：
 - trading_calendar：对齐交易日历；
 - ohlc_strategy：OHLC 缺失值处理；
 - volume_strategy：成交量缺失修复；
 - anomaly_strategy：异常数据识别；
 - quality_reporter：生成清洗质量报告。
- **策略抽象类：** Strategy 所有清洗策略统一通过 apply(df) 对数据进行处理，实现模块的可替换与可扩展。
- **OHLC 数据处理策略：**
 - ForwardFillOHLC：前向填充；
 - PrevCloseOHLC：使用上一日收盘价补齐完整 K 线；
 - InterpolateOHLC：基于插值的连续补全；
 - DropMissingOHLC：删除存在缺失的记录。
- **成交量处理策略：**
 - ZeroVolume：缺失量置零；
 - MeanVolume：用均值替代；
 - FFillVolume：前向填充方式平滑处理。

- 异常检测与质量报告：

- AnomalyDetector：标记价格或成交量异常波动；
- DataQualityReport：统计缺失率、异常率等质量指标。

整体流程为：对齐交易日历 → OHLC 修复 → 成交量修复 → 异常识别 → 数据质量评估。

6 结语

Qlib 通过统一的数据接口、基于表达式的特征构建体系、可插拔的模型与策略模块，以及可复现的实验管理机制，为量化研究提供了一条从数据到模型再到回测的标准化工作流。

本研究的拓展重点主要体现在以下几点：

- (1) 在数据层面，通过增强 QA/QC 模块来适配不同市场、不同频率甚至高频数据的清洗规范；
- (2) 在特征建模层面，针对特定策略目标构建具有解释性的因子体系，并支持可学习或结构化特征表达；
- (3) 在模型训练与评估阶段，结合实验管理与可复现机制，使模型迭代与性能比较更加系统化。

这些拓展内容的共同目标在于：将 Qlib 提供的“通用研究框架”与具体的量化研究需求相衔接，实现从需求分析 → 模型建构 → 实验验证的闭环流程，使研究过程具有可控性、可追踪性与可演化性。

未来工作可进一步沿以下方向深化：

- 针对行业、风格或市场状态变化，构建动态可适配的因子选择与模型调参机制；
- 推进从离线回测到实时部署的自动化衔接，使策略开发与迭代周期进一步缩短；
- 探索可解释性建模，以提升模型在策略风险控制与稳定收益中的实际可用性。

是为本次报告全部内容