In [1]:
```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler
import warnings
warnings.filterwarnings('ignore')

import yfinance as yf
from pathlib import Path


from scipy.stats import spearmanr
from sklearn.feature_selection import mutual_info_regression
```

In [2]:
```python
class OutputConfig:
    # 设置为True可以显示详细输出，False只显示关键信息
    VERBOSE = False
    SHOW_PROGRESS = True
    SHOW_FEATURE_DETAILS = False
    SHOW_TRAINING_DETAILS = False
```

数据读取

In [ ]:
```python
class DataLoader:

    def __init__(self, data_folder_path):
        self.data_folder = Path(data_folder_path)
        self.feature_data = None

    def load_parquet_files(self):
        parquet_files = list(self.data_folder.glob("*.parquet"))
        if not parquet_files:
            raise ValueError(f"在 {self.data_folder} 中没有找到parquet文件")

        features_dict = {}
        for file_path in parquet_files:
            feature_name = file_path.stem  # 获取文件名（不带扩展名）
            try:
                df = pd.read_parquet(file_path)
                # 使用特征名作为列名前缀，避免合并时的列名冲突
                df = df.add_prefix(f"{feature_name}_")
                features_dict[feature_name] = df
                print(f"成功加载特征: {feature_name}, 形状: {df.shape}")
            except Exception as e:
                print(f"加载 {feature_name} 失败: {e}")

        return self.merge_features(features_dict)

    def merge_features(self, features_dict):
        if not features_dict:
            raise ValueError("没有可用的特征数据")

        # 首先将所有DataFrame的索引统一为日期类型
        for name, df in features_dict.items():
            if df.index.name != 'date' and 'date' in df.columns:
                df = df.set_index('date')

            # 确保索引是日期类型
            df.index = pd.to_datetime(df.index)
            features_dict[name] = df.sort_index()

        # 使用concat而不是merge来避免列名冲突
        # 但需要确保所有DataFrame有相同的索引
        all_data = []
        for name, df in features_dict.items():
            all_data.append(df)

        # 使用outer连接合并所有数据
        merged_data = pd.concat(all_data, axis=1, join='outer')

        print(f"合并后数据形状: {merged_data.shape}")
        return merged_data

    def find_target_column(self, data):
        # 查找包含 'ret_21' 的列，不区分大小写
        target_cols = [col for col in data.columns if 'ret_21' in col.lower()]
        if target_cols:
            return target_cols[0]
        else:
            print("警告: 未找到目标变量列")
            return None
```

数据处理

In [ ]:
```python
class DataCleaner:

    def __init__(self, nan_threshold=0.8):
        self.nan_threshold = nan_threshold
```

```python
        self.kept_features = []

    def remove_high_nan_features(self, data):
        # 移除超过阈值NaN比例的特征
        if data.empty:
            return data

        nan_ratio = data.isnull().sum() / len(data)
        features_to_keep = nan_ratio[nan_ratio <= self.nan_threshold].index.tolist()

        # 记录被移除的特征
        removed_features = set(data.columns) - set(features_to_keep)

        self.kept_features = features_to_keep
        return data[features_to_keep]

    def safe_forward_fill(self, data):
        # 前向填充，处理开头为NaN的情况
        if data.empty:
            return data

        # 首先前向填充
        data_filled = data.ffill()

        # 检查是否还有NaN（出现在开头）
        if data_filled.isnull().any().any():
            print("检测到开头存在NaN，使用后向填充处理开头数据...")
            # 对于开头的NaN，使用后向填充（不会泄漏未来信息）
            data_filled = data_filled.bfill()

            # 如果还有NaN（全部为NaN的列），填充0
            if data_filled.isnull().any().any():
                print("使用0填充剩余的NaN值...")
                data_filled = data_filled.fillna(0)

        return data_filled

    def clean_data(self, data):
        print(f"原始数据形状: {data.shape}")

        # 移除高缺失率特征
        data_cleaned = self.remove_high_nan_features(data)
        print(f"移除高缺失率特征后形状: {data_cleaned.shape}")

        # 安全填充
        data_filled = self.safe_forward_fill(data_cleaned)
        print(f"填充后数据形状: {data_filled.shape}")

        # 检查是否还有NaN
        remaining_nans = data_filled.isnull().sum().sum()
        if remaining_nans > 0:
            print(f"警告: 数据中仍有 {remaining_nans} 个NaN值")

        return data_filled
```

宏观数据提取

In [ ]:

```python
class MacroDataEnhancer:

    def __init__(self):
        self.macro_features = []
        self.downloaded_data = {}

    def download_macro_data(self, start_date, end_date):
        macro_tickers = {
            'dollar_index': 'DX-Y.NYB',  # 美元指数
            'vix': '^VIX',               # VIX波动率指数
            'bond_yield_10y': '^TNX',    # 10年期国债收益率
            'sp500': '^GSPC',            # S&P 500指数
            'gold': 'GC=F',              # 黄金价格
            'oil': 'CL=F'                # 原油价格
        }

        for name, ticker in macro_tickers.items():
            try:
                print(f"正在下载 {name} 数据...")

                # 使用正确的yfinance下载方法
                import yfinance as yf

                # 下载数据
                data = yf.download(
                    ticker,
                    start=start_date,
                    end=end_date,
                    progress=False,
                    auto_adjust=True
                )

                if not data.empty:
                    # 使用调整后的收盘价
```

```python
                if 'Adj Close' in data.columns:
                    macro_series = data['Adj Close']
                else:
                    macro_series = data['Close']

                # 重命名序列
                macro_series.name = name
                self.downloaded_data[name] = macro_series
                print(f"成功下载 {name} 数据，共 {len(macro_series)} 个数据点")
            else:
                print(f"警告: {name} 数据为空")

        except Exception as e:
            print(f"下载 {name} 失败: {str(e)}")
            # 如果下载失败，创建一个空的Series作为占位符
            empty_series = pd.Series([], dtype=float, name=name)
            self.downloaded_data[name] = empty_series

    def add_all_macro_data(self, data, start_date, end_date):
        self.download_macro_data(start_date, end_date)

        enhanced_data = data.copy()
        macro_data_added = 0

        for macro_name, macro_series in self.downloaded_data.items():
            if len(macro_series) == 0:
                print(f"跳过空的宏观数据: {macro_name}")
                continue

            try:
                # 确保索引类型一致
                macro_series.index = pd.to_datetime(macro_series.index)
                enhanced_data.index = pd.to_datetime(enhanced_data.index)

                # 确保我们处理的是Series而不是DataFrame
                if isinstance(macro_series, pd.DataFrame):
                    print(f"警告: {macro_name} 是DataFrame而不是Series，尝试提取第一列")
                    macro_series = macro_series.iloc[:, 0]  # 取第一列
                    macro_series.name = macro_name

                # 将Series转换为DataFrame进行合并
                macro_df = pd.DataFrame({macro_name: macro_series})

                # 合并宏观数据
                enhanced_data = enhanced_data.merge(
                    macro_df,
                    left_index=True,
                    right_index=True,
                    how='left'
                )
                self.macro_features.append(macro_name)
                macro_data_added += 1
                print(f"已添加宏观特征: {macro_name}")

            except Exception as e:
                print(f"添加宏观特征 {macro_name} 失败: {str(e)}")

        print(f"宏观数据添加完成，成功添加 {macro_data_added} 个宏观特征")
        return enhanced_data
```

特征处理

初步特征筛选

In [ ]:
```python
class FeatureSelector:

    def __init__(self, max_features=1000, correlation_threshold=0.01,
                 mutual_info_threshold=0.01, variance_threshold=0.01):
        self.max_features = max_features
        self.correlation_threshold = correlation_threshold
        self.mutual_info_threshold = mutual_info_threshold
        self.variance_threshold = variance_threshold
        self.selected_features = []

    def calculate_feature_variance(self, data):
        variances = data.var()
        return variances

    def calculate_target_correlation(self, data, target_col='ret_21D'):
        correlations = {}
        target = data[target_col]

        for column in data.columns:
            if column != target_col:
                # Spearman相关系数，对异常值更稳健
                valid_data = data[[column, target_col]].dropna()
                if len(valid_data) > 10:
                    corr, _ = spearmanr(valid_data[column], valid_data[target_col])
                    correlations[column] = abs(corr) if not np.isnan(corr) else 0
```

```python
        return correlations

    def calculate_mutual_information(self, data, target_col='ret_21D', sample_fraction=0.1):
        # 抽样计算以节省时间
        sample_data = data.sample(frac=sample_fraction, random_state=42) if len(data) > 1000 else data
        sample_data = sample_data.dropna()

        if len(sample_data) < 50:
            return {}

        X_sample = sample_data.drop(target_col, axis=1)
        y_sample = sample_data[target_col]

        # 只计算部分特征以节省时间
        max_features_to_test = min(1000, len(X_sample.columns))
        features_to_test = np.random.choice(X_sample.columns, max_features_to_test, replace=False)

        mi_scores = {}
        for feature in features_to_test:
            try:
                mi = mutual_info_regression(
                    X_sample[[feature]].values.reshape(-1, 1),
                    y_sample,
                    random_state=42
                )[0]
                mi_scores[feature] = mi
            except:
                mi_scores[feature] = 0

        return mi_scores

    def select_features_static(self, data, target_col='ret_21D'):
        """静态特征选择"""
        print(f"原始特征数量: {len(data.columns)}")

        # 1. 移除低方差特征
        variances = self.calculate_feature_variance(data.drop(target_col, axis=1))
        high_variance_features = variances[variances > self.variance_threshold].index.tolist()
        print(f"高方差特征数量: {len(high_variance_features)}")

        # 2. 计算与目标的相关性
        correlations = self.calculate_target_correlation(data[high_variance_features + [target_col]], target_col)

        # 3. 计算互信息（选择性进行，因为计算成本较高）
        mi_scores = {}
        if len(high_variance_features) > 1000:
            print("计算互信息...")
            mi_scores = self.calculate_mutual_information(data[high_variance_features + [target_col]], target_col)

        # 4. 综合评分
        feature_scores = {}
        for feature in high_variance_features:
            corr_score = correlations.get(feature, 0)
            mi_score = mi_scores.get(feature, 0)

            # 综合评分：相关性权重0.7，互信息权重0.3
            combined_score = 0.7 * corr_score + 0.3 * mi_score
            feature_scores[feature] = combined_score

        # 5. 选择Top K特征
        sorted_features = sorted(feature_scores.items(), key=lambda x: x[1], reverse=True)
        selected_features = [feature for feature, score in sorted_features[:self.max_features]]

        # 确保目标变量在最终数据中
        final_features = selected_features + [target_col]

        print(f"静态特征选择完成，选择 {len(selected_features)} 个特征")
        print(f"Top 10 特征: {selected_features[:10]}")

        self.selected_features = selected_features
        return data[final_features]
```

特征工程

In [ ]:

```python
class FeatureProcessor:

    def __init__(self):
        pass

    def create_lag_features(self, data, lags=[1, 7, 21]):
        lagged_data = data.copy()

        # 排除目标变量
        original_columns = [col for col in data.columns if col != 'ret_21D']
        lag_features_created = 0

        for lag in lags:
            for column in original_columns:
                new_col_name = f'{column}_lag_{lag}'
                lagged_data[new_col_name] = data[column].shift(lag)
```

```python
                lag_features_created += 1

        print(f"滞后特征处理后数据形状: {lagged_data.shape}")
        return lagged_data

    def calculate_rolling_features(self, data, windows=[7, 21, 63]):
        """计算滚动统计特征"""
        print("计算滚动特征...")
        rolled_data = data.copy()

        # 排除目标变量
        original_columns = [col for col in data.columns if col != 'ret_21D']
        roll_features_created = 0

        for window in windows:
            for column in original_columns:
                # 滚动均值
                rolled_data[f'{column}_roll_mean_{window}'] = data[column].rolling(window, min_periods=1).mean()
                # 滚动标准差
                rolled_data[f'{column}_roll_std_{window}'] = data[column].rolling(window, min_periods=1).std()
                roll_features_created += 2

        print(f"创建了 {roll_features_created} 个滚动特征")
        print(f"滚动特征处理后数据形状: {rolled_data.shape}")
        return rolled_data
```

Testing

In [ ]:

```python
def run_real_data_pipeline():
    """使用真实数据运行数据处理流程 - 集成特征选择"""
    print("=" * 60)
    print("开始真实数据处理流程")
    print("=" * 60)

    # 1. 加载数据
    print("\n1. 加载数据...")
    data_folder_path = "/Users/tuibubansurfacepro/Desktop/flab ai/mnt/nas/yicheng/exercise_flab"
    data_loader = DataLoader(data_folder_path)

    try:
        main_data = data_loader.load_parquet_files()
        print(f"成功加载数据: {main_data.shape[0]} 行 × {main_data.shape[1]} 列")
        print(f"时间范围: {main_data.index.min().strftime('%Y-%m-%d')} 至 {main_data.index.max().strftime('%Y-%m-%d')}")

        # 查找目标变量
        target_col = data_loader.find_target_column(main_data)
        if target_col:
            main_data = main_data.rename(columns={target_col: 'ret_21D'})
            target_stats = main_data['ret_21D'].describe()
            print(f"目标变量: 均值={target_stats['mean']:.4f}, 标准差={target_stats['std']:.4f}")
        else:
            print("警告: 数据中未找到目标变量 'ret_21D'")

    except Exception as e:
        print(f"数据加载失败: {e}")
        return None, None, None

    # 2. 添加宏观数据
    print("\n2. 添加宏观数据...")
    try:
        macro_enhancer = MacroDataEnhancer()
        start_date = main_data.index.min().strftime('%Y-%m-%d')
        end_date = main_data.index.max().strftime('%Y-%m-%d')
        enhanced_data = macro_enhancer.add_all_macro_data(main_data, start_date, end_date)
        print(f"成功添加 {len(macro_enhancer.macro_features)} 个宏观特征")
    except Exception as e:
        print(f"宏观数据添加失败: {e}")
        enhanced_data = main_data.copy()

    # 3. 数据清洗
    print("\n3. 数据清洗...")
    try:
        cleaner = DataCleaner(nan_threshold=0.8)
        cleaned_data = cleaner.clean_data(enhanced_data)
        print(f"清洗完成: 保留 {len(cleaner.kept_features)} 个特征")
    except Exception as e:
        print(f"数据清洗失败: {e}")
        return None, None, None

    # 4. 静态特征选择
    print("\n4. 特征选择...")
    try:
        feature_selector = FeatureSelector(max_features=1000)
        selected_data = feature_selector.select_features_static(cleaned_data)
        print(f"特征选择完成: 从 {len(cleaned_data.columns)} 个特征中选择 {len(feature_selector.selected_features)} 个")
    except Exception as e:
        print(f"特征选择失败: {e}")
        selected_data = cleaned_data

    # 5. 特征工程
    print("\n5. 特征工程...")
```

```python
    try:
        processor = FeatureProcessor()

        # 创建滞后特征
        data_with_lags = processor.create_lag_features(selected_data, lags=[1, 5, 21])

        # 创建滚动特征
        data_with_features = processor.calculate_rolling_features(data_with_lags, windows=[5, 21, 63])

        print(f"特征工程完成")
        print(f"最终数据维度: {data_with_features.shape[0]} 样本 × {data_with_features.shape[1]} 特征")

    except Exception as e:
        print(f"特征工程失败: {e}")
        return None, None, None

    print("\n" + "=" * 60)
    print("✅ 数据处理流程完成!")
    print("=" * 60)

    return cleaned_data, data_with_features, processor
```

In [ ]:

```python
# 运行真实数据处理流程
if __name__ == "__main__":
    data_folder_path = "./flab ai/mnt/nas/yicheng/exercise_flab"

    print(f"使用数据路径: {data_folder_path}")

    cleaned_data, processed_data, feature_processor = run_real_data_pipeline()

    if processed_data is not None:
        print("\n数据处理结果摘要:")
        print(f"- 清洗后数据形状: {cleaned_data.shape if cleaned_data is not None else 'N/A'}")
        print(f"- 特征工程后数据形状: {processed_data.shape}")
        print(f"- 特征数量: {len(processed_data.columns)}")
        print(f"- 数据时间范围: {processed_data.index.min()} 到 {processed_data.index.max()}")

        # 检查目标变量
        if 'ret_21D' in processed_data.columns:
            target_data = processed_data['ret_21D'].dropna()
            print(f"- 目标变量有效样本数: {len(target_data)}")
            print(f"- 目标变量统计: 均值={target_data.mean():.6f}, 标准差={target_data.std():.6f}")

        # 保存处理后的数据 (可选)
        save_option = input("\n是否保存处理后的数据? (y/n): ").lower()
        if save_option == 'y':
            output_path = "./processed_data.parquet"
            processed_data.to_parquet(output_path)
            print(f"数据已保存至: {output_path}")
    else:
        print("\n数据处理失败, 请检查错误信息。")
```

Modeling

In [3]:

```python
import xgboost as xgb
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

动态特征筛选

In [4]:

```python
class DynamicFeatureOptimizer:
    """动态特征优化器 - 在在线学习过程中优化特征集"""

    def __init__(self, initial_features, max_features=500,
                 importance_threshold=0.001, stability_window=5):
        self.initial_features = initial_features
        self.max_features = max_features
        self.importance_threshold = importance_threshold
        self.stability_window = stability_window

        self.feature_importance_history = []
        self.current_feature_set = set(initial_features)
        self.feature_stability_count = {}

    def update_feature_set(self, feature_importance_df, current_date):
        """基于特征重要性更新特征集"""
        if feature_importance_df is None or len(feature_importance_df) == 0:
            return self.current_feature_set

        # 记录特征重要性
        self.feature_importance_history.append({
            'date': current_date,
            'importance': feature_importance_df.set_index('feature')['importance'].to_dict()
        })

        # 计算特征稳定性
        for feature in feature_importance_df['feature']:
            if feature in self.feature_stability_count:
                self.feature_stability_count[feature] += 1
```

```python
            else:
                self.feature_stability_count[feature] = 1

        # 选择重要且稳定的特征
        recent_importance = feature_importance_df.set_index('feature')['importance']
        # 过滤低重要性特征
        important_features = recent_importance[recent_importance > self.importance_threshold].index.tolist()

        # 优先选择稳定性高的特征
        feature_stability = pd.Series(self.feature_stability_count)
        stable_features = feature_stability[feature_stability >= self.stability_window].index.tolist()

        # 合并重要且稳定的特征
        candidate_features = set(important_features) & set(stable_features)

        # 如果候选特征太少，补充一些重要性高的特征
        if len(candidate_features) < self.max_features // 2:
            top_features = recent_importance.nlargest(self.max_features).index.tolist()
            candidate_features.update(top_features[:self.max_features // 2])

        # 限制特征数量
        if len(candidate_features) > self.max_features:
            # 按重要性排序并选择Top K
            candidate_importance = recent_importance.reindex(list(candidate_features)).fillna(0)
            top_candidates = candidate_importance.nlargest(self.max_features).index.tolist()
            self.current_feature_set = set(top_candidates)
        else:
            self.current_feature_set = candidate_features

        print(f"动态特征优化: 从 {len(feature_importance_df)} 个特征中选择 {len(self.current_feature_set)} 个特征")

        return self.current_feature_set
```

模型搭建

树模型预测

In [5]:
```python
class OnlineTreeModel:
    """在线学习树模型 - 集成特征优化"""

    def __init__(self, model_type='xgboost', model_params=None,
                 train_window=756, retrain_freq=42, prediction_horizon=21,
                 normalization_window=252, dynamic_feature_selection=True,
                 max_features=200):
        """
        初始化在线学习模型

        参数:
        - model_type: 模型类型
        - model_params: 模型参数
        - train_window: 训练窗口大小
        - retrain_freq: 重新训练频率
        - prediction_horizon: 预测horizon
        - normalization_window: 标准化窗口
        - dynamic_feature_selection: 是否启用动态特征选择
        - max_features: 最大特征数量
        """
        self.model_type = model_type
        self.model_params = model_params or {}
        self.train_window = train_window
        self.retrain_freq = retrain_freq
        self.prediction_horizon = prediction_horizon
        self.normalization_window = normalization_window
        self.dynamic_feature_selection = dynamic_feature_selection
        self.max_features = max_features

        self.model = None
        self.feature_importance_history = []
        self.prediction_history = []
        self.normalization_params = {}
        self.normalization_history = {}
        self.feature_optimizer = None
        self.current_feature_set = None

        # 设置默认模型参数
        if not self.model_params:
            if model_type == 'xgboost':
                self.model_params = {
                    'n_estimators': 50,
                    'max_depth': 6,
                    'learning_rate': 0.05,
                    'subsample': 0.7,
                    'colsample_bytree': 0.7,
                    'random_state': 42,
                    'n_jobs': -1
                }

    def initialize_model(self):
        """初始化模型"""
```

```python
        if self.model_type == 'xgboost':
            import xgboost as xgb
            self.model = xgb.XGBRegressor(**self.model_params)
        elif self.model_type == 'random_forest':
            from sklearn.ensemble import RandomForestRegressor
            self.model = RandomForestRegressor(**self.model_params)
        else:
            raise ValueError(f"不支持的模型类型: {self.model_type}")

    def calculate_mad(self, series):
        """计算平均绝对偏差 (MAD) - 替代已弃用的 .mad() 方法"""
        if len(series) == 0:
            return 0
        median = series.median()
        return (series - median).abs().mean()

    def calculate_normalization_params(self, data, current_date):
        """计算标准化参数 - 使用滑动窗口"""
        # 获取当前日期之前的标准化窗口数据
        historical_data = data[data.index <= current_date]
        if len(historical_data) < self.normalization_window:
            window_data = historical_data
        else:
            window_data = historical_data.tail(self.normalization_window)

        normalization_params = {}

        for column in data.columns:
            if column != 'ret_21D':
                window_values = window_data[column].dropna()

                if len(window_values) < 10:
                    continue

                mean_val = window_values.mean()
                std_val = window_values.std()

                if std_val < 1e-10:
                    median_val = window_values.median()
                    normalization_params[column] = {
                        'mean': median_val,
                        'std': 1.0,
                        'method': 'median_centering'
                    }
                else:
                    # 使用自定义的MAD计算方法
                    mad_val = self.calculate_mad(window_values)

                    # 检查异常大的标准差 (可能由于异常值)
                    if std_val > 10 * mad_val:  # 用MAD检测异常
                        # 使用更稳健的标准化
                        median_val = window_values.median()
                        normalization_params[column] = {
                            'mean': median_val,
                            'std': mad_val if mad_val > 1e-10 else 1.0,
                            'method': 'robust_normalization'
                        }
                    else:
                        # 正常标准化
                        normalization_params[column] = {
                            'mean': mean_val,
                            'std': std_val,
                            'method': 'standard_normalization'
                        }

                if column not in self.normalization_history:
                    self.normalization_history[column] = []

                self.normalization_history[column].append({
                    'date': current_date,
                    'mean': normalization_params[column]['mean'],
                    'std': normalization_params[column]['std'],
                    'method': normalization_params[column]['method']
                })

        return normalization_params

    def apply_normalization(self, data, normalization_params):
        """应用标准化"""
        normalized_data = data.copy()

        for column, params in normalization_params.items():
            if column in normalized_data.columns:
                mean_val = params['mean']
                std_val = params['std']

                if std_val > 1e-10:
                    normalized_data[column] = (data[column] - mean_val) / std_val
                else:
                    normalized_data[column] = data[column] - mean_val

        return normalized_data
```

```python
    def prepare_training_data(self, data, current_date, initial_training=False):
        """准备训练数据 - 集成特征选择"""
        # 获取当前日期之前的数据
        historical_data = data[data.index <= current_date]

        if len(historical_data) < self.train_window:
            train_data = historical_data
        else:
            train_data = historical_data.tail(self.train_window)

        # 移除目标变量为NaN的样本
        train_data = train_data.dropna(subset=['ret_21D'])

        if len(train_data) < 100:
            return None, None, None

        # 分离特征和目标
        X = train_data.drop('ret_21D', axis=1)
        y = train_data['ret_21D']

        # 初始训练或首次训练时初始化特征优化器
        if initial_training or self.feature_optimizer is None:
            if self.dynamic_feature_selection:
                self.feature_optimizer = DynamicFeatureOptimizer(
                    initial_features=X.columns.tolist(),
                    max_features=self.max_features
                )
                self.current_feature_set = set(X.columns.tolist())

        # 动态特征选择
        if self.dynamic_feature_selection and self.feature_optimizer and not initial_training:
            # 使用当前特征集
            available_features = set(X.columns) & self.current_feature_set
            if available_features:
                X = X[list(available_features)]

        # 移除在训练集中全为NaN的列
        X = X.dropna(axis=1, how='all')

        # 移除常数特征
        constant_cols = [col for col in X.columns if X[col].nunique() <= 1]
        X = X.drop(columns=constant_cols)

        if len(X.columns) == 0:
            return None, None, None

        # 计算标准化参数
        self.normalization_params = self.calculate_normalization_params(X, current_date)

        # 应用标准化
        X_normalized = self.apply_normalization(X, self.normalization_params)

        # 前向填充剩余的NaN
        X_normalized = X_normalized.ffill().bfill().fillna(0)

        return X_normalized, y, X_normalized.columns.tolist()

    def train_model(self, data, current_date, initial_training=False):
        """训练模型 - 集成动态特征优化"""
        X, y, feature_names = self.prepare_training_data(data, current_date, initial_training)

        if X is None or len(X) == 0:
            if OutputConfig.SHOW_TRAINING_DETAILS:
                print(f"在 {current_date.strftime('%Y-%m-%d')} 训练数据不足，跳过训练")
            return False

        try:
            if self.model is None:
                self.initialize_model()

            self.model.fit(X, y)

            # 记录特征重要性
            if hasattr(self.model, 'feature_importances_'):
                importance_df = pd.DataFrame({
                    'feature': feature_names,
                    'importance': self.model.feature_importances_,
                    'date': current_date
                }).sort_values('importance', ascending=False)

                self.feature_importance_history.append(importance_df)

                # 动态特征优化
                if self.dynamic_feature_selection and self.feature_optimizer and not initial_training:
                    self.current_feature_set = self.feature_optimizer.update_feature_set(
                        importance_df, current_date
                    )

            if initial_training or OutputConfig.SHOW_TRAINING_DETAILS:
                print(f"{current_date.strftime('%Y-%m-%d')}: 训练完成 - {len(X)} 样本, {len(feature_names)} 特征")

            return True
```

```python
        except Exception as e:
            if OutputConfig.SHOW_TRAINING_DETAILS:
                print(f"{current_date.strftime('%Y-%m-%d')}: 训练失败 - {e}")
            return False

    def predict(self, data, current_date):
        """预测当前日期的未来21天收益率"""
        if self.model is None:
            return None

        try:
            # 获取当前日期的特征
            current_features = data[data.index == current_date].drop('ret_21D', axis=1)

            if len(current_features) == 0:
                return None

            # 动态特征选择
            if self.dynamic_feature_selection and self.current_feature_set:
                available_features = set(current_features.columns) & self.current_feature_set
                current_features = current_features[list(available_features)]

            # 应用相同的标准化
            current_features_normalized = self.apply_normalization(current_features, self.normalization_params)

            # 确保特征与训练时一致
            if hasattr(self.model, 'feature_names_in_'):
                expected_features = self.model.feature_names_in_
                missing_features = set(expected_features) - set(current_features_normalized.columns)

                if missing_features:
                    for feature in missing_features:
                        current_features_normalized[feature] = 0

                current_features_normalized = current_features_normalized[expected_features]

            # 处理NaN值
            current_features_normalized = current_features_normalized.ffill().bfill().fillna(0)

            prediction = self.model.predict(current_features_normalized)[0]

            # 记录预测
            actual_return = None
            if 'ret_21D' in data.columns:
                actual_data = data.loc[data.index == current_date, 'ret_21D']
                if len(actual_data) > 0:
                    actual_return = actual_data.iloc[0]

            self.prediction_history.append({
                'date': current_date,
                'prediction': prediction,
                'actual': actual_return
            })

            return prediction

        except Exception as e:
            print(f"在 {current_date} 预测失败: {e}")
            return None
```

信号搭建

In [6]:
```python
class AdvancedPortfolioManager:

    def __init__(self, initial_capital=1000000, max_position=0.02,
                 transaction_cost=0.005, volatility_lookback=126,
                 kelly_fraction=0.08, min_volatility=0.03,
                 prediction_threshold=0.01):
        """
        统一的投资组合管理器 - 保守参数
        """
        self.initial_capital = initial_capital
        self.current_capital = initial_capital
        self.max_position = max_position
        self.transaction_cost = transaction_cost
        self.volatility_lookback = volatility_lookback
        self.kelly_fraction = kelly_fraction
        self.min_volatility = min_volatility
        self.prediction_threshold = prediction_threshold

        # 投资组合跟踪
        self.portfolio_value = [initial_capital]
        self.dates = []
        self.positions = {}
        self.trade_history = []
        self.portfolio_weights_history = []

        # 性能监控
        self.consecutive_losses = 0
        self.max_consecutive_losses = 5
```

```python
    def calculate_volatility(self, returns_series, lookback=None):
        """统一的波动率计算方法"""
        if lookback is None:
            lookback = self.volatility_lookback

        if len(returns_series) < 30:
            return self.min_volatility

        available_data = returns_series.tail(min(lookback, len(returns_series))).dropna()

        if len(available_data) < 30:
            return self.min_volatility

        # 使用稳健的波动率估计
        median = available_data.median()
        mad = (available_data - median).abs().median()
        volatility = mad * 1.4826  # 将MAD转换为标准差估计

        # 严格的波动率限制
        volatility = max(volatility, self.min_volatility)
        volatility = min(volatility, 0.30)

        return volatility

    def conservative_kelly_sizing(self, prediction, volatility, recent_performance=None):
        """保守的凯利仓位管理"""
        if recent_performance is None:
            recent_performance = {'consecutive_losses': 0, 'win_rate': 0.5}

        # 更高的预测阈值
        if abs(prediction) < self.prediction_threshold:
            return 0

        # 连续亏损惩罚
        performance_penalty = 1.0
        if recent_performance.get('consecutive_losses', 0) > 2:
            performance_penalty = 0.5
        elif recent_performance.get('win_rate', 0.5) < 0.4:
            performance_penalty = 0.7

        # 基础凯利计算
        raw_kelly = prediction / (volatility ** 2)

        # 多重保守调整
        signal_strength = min(abs(prediction) / 0.08, 1.0)
        vol_penalty = 1.0 / (1.0 + 3.0 * volatility)

        adjusted_kelly = raw_kelly * signal_strength * vol_penalty * self.kelly_fraction * performance_penalty

        # 非常严格的仓位限制
        position_size = np.clip(adjusted_kelly, -self.max_position, self.max_position)

        # 更高的最小仓位阈值
        if abs(position_size) < 0.002:
            position_size = 0

        return position_size

    def get_recent_performance(self):
        """获取近期表现"""
        if len(self.trade_history) < 10:
            return {'consecutive_losses': self.consecutive_losses, 'win_rate': 0.5}

        recent_trades = self.trade_history[-10:]
        wins = sum(1 for trade in recent_trades if trade.get('portfolio_return', 0) > 0)
        win_rate = wins / len(recent_trades)

        return {
            'consecutive_losses': self.consecutive_losses,
            'win_rate': win_rate
        }

    def execute_advanced_trades(self, date, asset_data, predictions, current_capital):
        """执行高级交易 - 统一方法名"""
        self.current_capital = current_capital

        if not predictions:
            self.portfolio_value.append(current_capital)
            self.dates.append(date)
            return current_capital, 0, {}

        asset_name = list(predictions.keys())[0] if predictions else 'primary_asset'
        prediction = predictions.get(asset_name, 0)

        # 计算近期表现
        recent_performance = self.get_recent_performance()

        # 波动率计算
        volatility = self.min_volatility
        if asset_name in asset_data and 'returns' in asset_data[asset_name]:
            returns_series = asset_data[asset_name]['returns']
            volatility = self.calculate_volatility(returns_series)
```

```python
        # 保守仓位计算
        position_size = self.conservative_kelly_sizing(prediction, volatility, recent_performance)

        if position_size == 0:
            self.portfolio_value.append(current_capital)
            self.dates.append(date)
            return current_capital, 0, {}

        # 交易成本
        transaction_cost = abs(position_size) * self.transaction_cost

        # 收益计算 - 使用更保守的方法
        portfolio_return = 0
        if asset_name in asset_data and 'returns' in asset_data[asset_name]:
            returns_series = asset_data[asset_name]['returns']
            if len(returns_series) > 0:
                # 使用最近5天的平均收益率
                recent_returns = returns_series.tail(5)
                asset_return = recent_returns.mean() if len(recent_returns) > 0 else 0
                portfolio_return = position_size * asset_return

        # 扣除交易成本
        portfolio_return -= transaction_cost

        # 更新资金
        new_capital = current_capital * (1 + portfolio_return)
        self.current_capital = new_capital

        # 记录交易
        weights = {asset_name: position_size}
        trade_record = {
            'date': date,
            'weights': weights,
            'portfolio_return': portfolio_return,
            'transaction_cost': transaction_cost,
            'capital_before': current_capital,
            'capital_after': new_capital,
            'prediction': prediction
        }
        self.trade_history.append(trade_record)
        self.portfolio_value.append(new_capital)
        self.dates.append(date)
        self.portfolio_weights_history.append(trade_record)

        # 更新连续亏损计数
        if portfolio_return < 0:
            self.consecutive_losses += 1
        else:
            self.consecutive_losses = 0

        # 如果连续亏损过多，输出警告
        if self.consecutive_losses >= self.max_consecutive_losses:
            print(f"警告: 连续{self.consecutive_losses}次亏损")

        return new_capital, portfolio_return, weights

    def get_performance_summary(self):
        """获取投资组合绩效摘要"""
        if len(self.portfolio_value) < 2:
            return {}

        portfolio_returns = pd.Series(self.portfolio_value).pct_change().dropna()

        total_return = (self.portfolio_value[-1] / self.portfolio_value[0] - 1) * 100
        annual_return = portfolio_returns.mean() * 252 * 100
        annual_volatility = portfolio_returns.std() * np.sqrt(252) * 100
        sharpe_ratio = annual_return / annual_volatility if annual_volatility > 0 else 0

        # 计算最大回撤
        portfolio_series = pd.Series(self.portfolio_value)
        rolling_max = portfolio_series.expanding().max()
        drawdowns = (portfolio_series - rolling_max) / rolling_max
        max_drawdown = drawdowns.min() * 100

        # 计算胜率
        winning_periods = len([r for r in portfolio_returns if r > 0])
        win_rate = winning_periods / len(portfolio_returns) * 100 if len(portfolio_returns) > 0 else 0

        return {
            'Total Return (%)': total_return,
            'Annual Return (%)': annual_return,
            'Annual Volatility (%)': annual_volatility,
            'Sharpe Ratio': sharpe_ratio,
            'Max Drawdown (%)': max_drawdown,
            'Win Rate (%)': win_rate,
            'Final Capital': self.portfolio_value[-1],
            'Number of Trades': len(self.trade_history)
        }
```

回测流程

In [7]:
```python
class EnhancedStrategyBacktester:

    def __init__(self, model, portfolio_manager):
        self.model = model
        self.portfolio_manager = portfolio_manager
        self.performance_metrics = {}

    def run_enhanced_backtest(self, data, start_date, end_date):
        """运行增强回测"""
        print("=" * 60)
        print("开始策略回测")
        print("=" * 60)

        # 筛选回测期间的数据
        backtest_data = data[(data.index >= start_date) & (data.index <= end_date)]

        # 确保dates变量被正确赋值
        if backtest_data.empty:
            print("警告: 回测期间没有数据!")
            return {
                'portfolio_values': [self.portfolio_manager.initial_capital],
                'portfolio_dates': [start_date],
                'weights_history': [],
                'predictions': [],
                'actual_returns': [],
                'signal_dates': [],
                'metrics': {},
                'feature_importance': self.model.feature_importance_history
            }

        dates = backtest_data.index.unique()
        dates = sorted(dates)

        capital = self.portfolio_manager.initial_capital
        portfolio_values = [capital]
        portfolio_dates = [dates[0] if len(dates) > 0 else start_date]
        portfolio_weights_history = []
        predictions_list = []
        actual_returns_list = []
        signal_dates = []

        print(f"回测期间: {start_date.strftime('%Y-%m-%d')} 至 {end_date.strftime('%Y-%m-%d')}")
        print(f"总交易日: {len(dates)}")

        for i, current_date in enumerate(dates):
            # 显示进度
            if OutputConfig.SHOW_PROGRESS and i % max(1, len(dates) // 20) == 0:
                progress = (i + 1) / len(dates) * 100
                print(f"进度: {i+1}/{len(dates)} ({progress:.1f}%)")

            # 定期重新训练模型
            if i % self.model.retrain_freq == 0:
                if OutputConfig.SHOW_TRAINING_DETAILS:
                    print(f"{current_date.strftime('%Y-%m-%d')}: 重新训练模型...")
                success = self.model.train_model(data, current_date)
                if not success and i > 0 and OutputConfig.SHOW_TRAINING_DETAILS:
                    print("训练失败, 使用之前的模型继续预测")

            # 进行预测
            prediction = self.model.predict(data, current_date)  # 使用完整data而不是backtest_data

            if prediction is not None:
                # 添加预测值调试
                if OutputConfig.VERBOSE:
                    print(f"{current_date.strftime('%Y-%m-%d')}: 预测值 = {prediction:.6f}")

                predictions_list.append(prediction)
                signal_dates.append(current_date)

                # 获取实际收益率
                actual_return_data = data.loc[data.index == current_date, 'ret_21D']
                if len(actual_return_data) > 0:
                    actual_return = actual_return_data.iloc[0]
                    actual_returns_list.append(actual_return)
                else:
                    actual_returns_list.append(0)

                # 构建资产数据
                asset_data = self.prepare_asset_data(data, current_date)

                # 执行高级交易
                try:
                    capital, portfolio_return, weights = self.portfolio_manager.execute_advanced_trades(
                        current_date, asset_data, {'primary_asset': prediction}, capital
                    )

                    # 记录交易结果
                    portfolio_values.append(capital)
                    portfolio_dates.append(current_date)
                    portfolio_weights_history.append({
                        'date': current_date,
                        'weights': weights,
```

```
                    'portfolio_return': portfolio_return,
                    'prediction': prediction,
                    'actual_return': actual_returns_list[-1] if actual_returns_list else None
                })

                # 显示交易结果（如果交易成功）
                if weights and any(w != 0 for w in weights.values()):
                    if OutputConfig.VERBOSE:
                        print(f"{current_date.strftime('%Y-%m-%d')}: 仓位 = {weights}, 收益 = {portfolio_return:.4f}")

            except Exception as e:
                print(f"  {current_date.strftime('%Y-%m-%d')}: 交易执行失败 - {e}")
                # 即使交易失败，也记录投资组合价值
                portfolio_values.append(capital)
                portfolio_dates.append(current_date)

        # 计算绩效指标
        self.performance_metrics = self.calculate_enhanced_metrics(
            portfolio_values, portfolio_weights_history, predictions_list, actual_returns_list
        )

        print(f"回测完成: {len(predictions_list)} 次预测, {len(portfolio_weights_history)} 次交易")

        return {
            'portfolio_values': portfolio_values,
            'portfolio_dates': portfolio_dates,
            'weights_history': portfolio_weights_history,
            'predictions': predictions_list,
            'actual_returns': actual_returns_list,
            'signal_dates': signal_dates,
            'metrics': self.performance_metrics,
            'feature_importance': self.model.feature_importance_history
        }


    def prepare_asset_data(self, data, current_date):
        """准备资产数据 - 为单一资产情况设计"""
        # 获取历史数据用于计算波动率等指标
        historical_data = data[data.index <= current_date]

        # 假设只有一个主要资产
        asset_data = {
            'primary_asset': {
                'returns': historical_data['ret_21D'],
                # 如果没有价格数据，用累积收益率模拟价格序列
                'prices': self.calculate_price_series(historical_data['ret_21D']),
                'volatility': self.calculate_rolling_volatility(historical_data['ret_21D'])
            }
        }

        return asset_data

    def calculate_price_series(self, returns_series, initial_price=100):
        """根据收益率序列计算价格序列"""
        if len(returns_series) == 0:
            return pd.Series([initial_price])

        # 计算累积收益率
        cumulative_returns = (1 + returns_series).cumprod()

        # 转换为价格序列
        price_series = initial_price * cumulative_returns

        return price_series

    def calculate_rolling_volatility(self, returns_series, window=63):
        """计算滚动波动率"""
        if len(returns_series) < window:
            return returns_series.std() if len(returns_series) > 0 else 0.02

        return returns_series.rolling(window=window, min_periods=10).std().iloc[-1] if len(returns_series) > 0 else 0.02

    def calculate_enhanced_metrics(self, portfolio_values, weights_history, predictions, actual_returns):
        """计算增强绩效指标"""
        portfolio_returns = pd.Series(portfolio_values).pct_change().dropna()

        if len(portfolio_returns) == 0:
            return {}

        # 基本指标
        total_return = (portfolio_values[-1] / portfolio_values[0] - 1) * 100
        annual_return = portfolio_returns.mean() * 252 * 100
        annual_volatility = portfolio_returns.std() * np.sqrt(252) * 100
        sharpe_ratio = annual_return / annual_volatility if annual_volatility > 0 else 0

        # 风险调整指标
        downside_returns = portfolio_returns[portfolio_returns < 0]
        sortino_ratio = annual_return / (downside_returns.std() * np.sqrt(252) * 100) if len(downside_returns) > 0 else 0

        # 最大回撤
        max_drawdown = self.calculate_max_drawdown(portfolio_values) * 100

        # Calmar比率
```

```
    calmar_ratio = annual_return / abs(max_drawdown) if max_drawdown != 0 else 0

    # 胜率
    winning_periods = len([r for r in portfolio_returns if r > 0])
    win_rate = winning_periods / len(portfolio_returns) if len(portfolio_returns) > 0 else 0

    # 预测精度指标
    prediction_accuracy = 0
    if len(predictions) > 0 and len(actual_returns) > 0:
        min_len = min(len(predictions), len(actual_returns))
        predictions_arr = np.array(predictions[:min_len])
        actual_arr = np.array(actual_returns[:min_len])

        # 相关性
        if min_len > 1:
            correlation_matrix = np.corrcoef(predictions_arr, actual_arr)
            correlation = correlation_matrix[0, 1] if not np.isnan(correlation_matrix[0, 1]) else 0
        else:
            correlation = 0

        # 均方误差
        mse = mean_squared_error(actual_arr, predictions_arr) if min_len > 0 else 0

        # 方向准确性
        direction_correct = np.sum(
            (predictions_arr > 0) == (actual_arr > 0)
        ) / min_len if min_len > 0 else 0

        # 信息系数 (IC)
        ic = correlation
    else:
        correlation = 0
        mse = 0
        direction_correct = 0
        ic = 0

    # 交易相关指标
    total_trades = len(weights_history)
    positive_trades = len([w for w in weights_history if w.get('portfolio_return', 0) > 0])
    trade_win_rate = positive_trades / total_trades if total_trades > 0 else 0

    # 平均持仓比例
    avg_position_size = np.mean([sum(w['weights'].values()) for w in weights_history]) if weights_history else 0

    metrics = {
        # 收益指标
        'Total Return (%)': total_return,
        'Annual Return (%)': annual_return,
        'Annual Volatility (%)': annual_volatility,
        'Sharpe Ratio': sharpe_ratio,
        'Sortino Ratio': sortino_ratio,
        'Calmar Ratio': calmar_ratio,
        'Max Drawdown (%)': max_drawdown,
        'Win Rate (%)': win_rate,

        # 预测精度指标
        'Prediction Correlation': correlation,
        'Information Coefficient': ic,
        'Prediction MSE': mse,
        'Direction Accuracy': direction_correct,

        # 交易指标
        'Number of Trades': total_trades,
        'Trade Win Rate (%)': trade_win_rate * 100,
        'Average Position Size (%)': avg_position_size * 100,

        # 其他
        'Final Portfolio Value': portfolio_values[-1],
        'Initial Capital': self.portfolio_manager.initial_capital
    }

    return metrics

def calculate_max_drawdown(self, portfolio_values):
    """计算最大回撤"""
    portfolio_series = pd.Series(portfolio_values)
    rolling_max = portfolio_series.expanding().max()
    drawdown = (portfolio_series - rolling_max) / rolling_max
    return drawdown.min()

def generate_enhanced_report(self, backtest_results):
    """生成增强回测报告"""
    print("\n" + "=" * 60)
    print("回测结果报告")
    print("=" * 60)

    metrics = backtest_results['metrics']

    print("\n核心绩效指标:")
    print("-" * 40)

    # 收益指标
    print("\n收益表现:")
```

```
        print(f"    总收益率: {metrics.get('Total Return (%)', 0):.2f}%")
        print(f"    年化收益率: {metrics.get('Annual Return (%)', 0):.2f}%")
        print(f"    年化波动率: {metrics.get('Annual Volatility (%)', 0):.2f}%")
        print(f"    夏普比率: {metrics.get('Sharpe Ratio', 0):.4f}")

        # 风险指标
        print("\n风险控制:")
        print(f"    最大回撤: {metrics.get('Max Drawdown (%)', 0):.2f}%")
        print(f"    索提诺比率: {metrics.get('Sortino Ratio', 0):.4f}")

        # 预测精度
        print("\n预测质量:")
        print(f"    预测相关性: {metrics.get('Prediction Correlation', 0):.4f}")
        print(f"    方向准确率: {metrics.get('Direction Accuracy', 0):.4f}")

        # 交易统计
        print("\n交易统计:")
        print(f"    交易次数: {metrics.get('Number of Trades', 0)}")
        print(f"    交易胜率: {metrics.get('Trade Win Rate (%)', 0):.2f}%")

        # 资金信息
        print("\n资金信息:")
        print(f"    初始资本: ${metrics.get('Initial Capital', 0):,.0f}")
        print(f"    最终价值: ${metrics.get('Final Portfolio Value', 0):,.0f}")
        print(f"    绝对收益: ${metrics.get('Final Portfolio Value', 0) - metrics.get('Initial Capital', 0):,.0f}")

        # 显示最重要的特征
        if backtest_results['feature_importance'] and OutputConfig.SHOW_FEATURE_DETAILS:
            latest_importance = backtest_results['feature_importance'][-1]
            top_features = latest_importance.head(5)
            print(f"\nTop 5 重要特征:")
            for i, (_, row) in enumerate(top_features.iterrows(), 1):
                print(f"    {i}. {row['feature'][:50]}...: {row['importance']:.4f}")


    def analyze_prediction_quality(self, backtest_results):
        """分析预测质量"""
        predictions = backtest_results['predictions']
        actual_returns = backtest_results['actual_returns']

        if len(predictions) == 0 or len(actual_returns) == 0:
            print("没有足够的预测数据进行分析")
            return

        min_len = min(len(predictions), len(actual_returns))
        predictions_arr = np.array(predictions[:min_len])
        actual_arr = np.array(actual_returns[:min_len])

        print("\n" + "=" * 60)
        print("预测质量分析")
        print("=" * 60)

        # 基本统计
        print(f"预测值统计:")
        print(f"  均值: {predictions_arr.mean():.6f}")
        print(f"  标准差: {predictions_arr.std():.6f}")
        print(f"  最小值: {predictions_arr.min():.6f}")
        print(f"  最大值: {predictions_arr.max():.6f}")

        print(f"\n实际值统计:")
        print(f"  均值: {actual_arr.mean():.6f}")
        print(f"  标准差: {actual_arr.std():.6f}")
        print(f"  最小值: {actual_arr.min():.6f}")
        print(f"  最大值: {actual_arr.max():.6f}")

        # 分位数分析
        prediction_quantiles = np.percentile(predictions_arr, [25, 50, 75])
        actual_quantiles = np.percentile(actual_arr, [25, 50, 75])

        print(f"\n分位数分析:")
        print(f"  预测值 - 25%: {prediction_quantiles[0]:.6f}, 中位数: {prediction_quantiles[1]:.6f}, 75%: {prediction_quantiles[
        print(f"  实际值 - 25%: {actual_quantiles[0]:.6f}, 中位数: {actual_quantiles[1]:.6f}, 75%: {actual_quantiles[2]:.6f}")

        # 信号强度分析
        strong_buy_signals = np.sum(predictions_arr > 0.05)  # 强买入信号
        strong_sell_signals = np.sum(predictions_arr < -0.05)  # 强卖出信号

        print(f"\n信号强度分析:")
        print(f"  强买入信号次数: {strong_buy_signals} ({strong_buy_signals/len(predictions_arr)*100:.1f}%)")
        print(f"  强卖出信号次数: {strong_sell_signals} ({strong_sell_signals/len(predictions_arr)*100:.1f}%)")

        # 保存预测质量分析
        prediction_analysis = {
            'predictions_mean': predictions_arr.mean(),
            'predictions_std': predictions_arr.std(),
            'actual_mean': actual_arr.mean(),
            'actual_std': actual_arr.std(),
            'correlation': np.corrcoef(predictions_arr, actual_arr)[0, 1] if len(predictions_arr) > 1 else 0,
            'direction_accuracy': np.sum((predictions_arr > 0) == (actual_arr > 0)) / len(predictions_arr)
        }

        pd.DataFrame([prediction_analysis]).to_csv("./prediction_quality_analysis.csv", index=False)
        print("预测质量分析已保存至: ./prediction_quality_analysis.csv")
```

Testing

In [8]:
```python
def main_model_training():
    print("=" * 80)
    print("开始在线学习模型训练和回测流程")
    print("=" * 80)

    # 1. 加载处理好的数据
    print("\n1. 加载数据...")
    try:
        processed_data_path = "./processed_data.parquet"
        processed_data = pd.read_parquet(processed_data_path)
        print(f"数据维度: {processed_data.shape[0]} × {processed_data.shape[1]}")
        print(f"时间范围: {processed_data.index.min().strftime('%Y-%m-%d')} 至 {processed_data.index.max().strftime('%Y-%m-%d')}"

        if 'ret_21D' in processed_data.columns:
            target_data = processed_data['ret_21D'].dropna()
            print(f"目标变量: {len(target_data)} 个有效样本")

    except Exception as e:
        print(f"加载数据失败: {e}")
        return

    # 2. 运行在线学习策略
    print("\n2. 模型训练与回测...")
    backtest_results, model, backtester = run_online_learning_strategy(processed_data)

    if backtest_results is not None:
        # 3. 保存结果和分析
        print("\n3. 结果保存...")
        save_and_analyze_results(backtest_results, model, backtester)

        # 显示关键结果摘要
        print("\n" + "=" * 60)
        print("流程完成摘要")
        print("=" * 60)

        metrics = backtest_results['metrics']
        print(f"最终绩效:")
        print(f"总收益率: {metrics.get('Total Return (%)', 0):.2f}%")
        print(f"年化收益率: {metrics.get('Annual Return (%)', 0):.2f}%")
        print(f"夏普比率: {metrics.get('Sharpe Ratio', 0):.4f}")
        print(f"最大回撤: {metrics.get('Max Drawdown (%)', 0):.2f}%")
        print(f"预测准确率: {metrics.get('Direction Accuracy', 0):.4f}")

        print(f"\n交易统计:")
        print(f"交易次数: {metrics.get('Number of Trades', 0)}")
        print(f"交易胜率: {metrics.get('Trade Win Rate (%)', 0):.2f}%")

        print(f"\n资金变化:")
        initial = metrics.get('Initial Capital', 0)
        final = metrics.get('Final Portfolio Value', 0)
        print(f"初始: ${initial:,.0f}")
        print(f"最终: ${final:,.0f}")
        print(f"收益: ${final - initial:,.0f}")

        print("\n" + "=" * 80)
        print("所有流程完成!")
        print("=" * 80)

    else:
        print("模型训练失败")

def run_online_learning_strategy(processed_data):
    """运行在线学习策略"""
    print("开始在线学习树模型策略")

    # 1. 初始化在线学习模型
    print("\n1. 初始化在线学习模型...")
    online_model = OnlineTreeModel(
        model_type='xgboost',
        model_params={
            'n_estimators': 50,  # 减少树的数量
            'max_depth': 6,      # 降低树深度
            'learning_rate': 0.05,  # 降低学习率
            'subsample': 0.7,
            'colsample_bytree': 0.7,
            'random_state': 42,
            'n_jobs': -1
        },
        train_window=756,  # 增加训练窗口到3年
        retrain_freq=42,   # 降低重新训练频率
        prediction_horizon=21,
        dynamic_feature_selection=True,
        max_features=200   # 减少特征数量
    )

    # 2. 初始化投资组合管理器
    print("2. 初始化投资组合管理器...")
    portfolio_manager = AdvancedPortfolioManager(
        initial_capital=1000000,
```

```python
        max_position=0.02,      # 2%最大仓位
        transaction_cost=0.005, # 0.5%交易成本
        kelly_fraction=0.08,    # 8%凯利分数
        min_volatility=0.03,    # 3%最小波动率
        prediction_threshold=0.01  # 1%预测阈值
    )

    # 3. 初始化回测器
    print("3. 初始化策略回测器...")
    backtester = EnhancedStrategyBacktester(online_model, portfolio_manager)

    # 4. 运行回测
    print("4. 运行回测...")
    # 使用后70%的数据进行回测，前20%用于初始训练
    split_idx = int(len(processed_data) * 0.3)
    start_date = processed_data.index[split_idx]
    end_date = processed_data.index[-1]

    print(f"回测期间: {start_date} 到 {end_date}")
    print(f"回测数据量: {len(processed_data[processed_data.index >= start_date])} 个交易日")

    # 初始训练
    print("进行初始模型训练...")
    initial_success = online_model.train_model(processed_data, start_date, initial_training=True)
    if not initial_success:
        print("初始训练失败，调整参数重试...")
        # 如果初始训练失败，尝试使用更大的窗口
        online_model.train_window = min(online_model.train_window, len(processed_data) // 2)
        initial_success = online_model.train_model(processed_data, start_date, initial_training=True)

    if initial_success:
        backtest_results = backtester.run_enhanced_backtest(processed_data, start_date, end_date)

        # 生成报告
        print("\n5. 生成回测报告...")
        backtester.generate_enhanced_report(backtest_results)

        return backtest_results, online_model, backtester
    else:
        print("初始训练失败，无法进行回测")
        return None, None, None

def save_and_analyze_results(backtest_results, model, backtester):
    """保存结果并进行详细分析"""

    # 1. 保存预测历史
    predictions_df = pd.DataFrame(model.prediction_history)
    predictions_df.to_csv("./prediction_history.csv", index=False)
    print("预测历史已保存至: ./prediction_history.csv")

    # 2. 保存特征重要性
    if model.feature_importance_history:
        feature_importance_df = pd.concat(model.feature_importance_history)
        feature_importance_df.to_csv("./feature_importance_history.csv", index=False)
        print("特征重要性历史已保存至: ./feature_importance_history.csv")

        # 分析特征重要性稳定性
        analyze_feature_stability(feature_importance_df)

    # 3. 保存投资组合结果
    portfolio_results = pd.DataFrame({
        'date': backtest_results['portfolio_dates'],
        'portfolio_value': backtest_results['portfolio_values']
    })
    portfolio_results.to_csv("./portfolio_results.csv", index=False)
    print("✓ 投资组合结果已保存至: ./portfolio_results.csv")

    # 4. 保存绩效指标
    metrics_df = pd.DataFrame([backtest_results['metrics']])
    metrics_df.to_csv("./performance_metrics.csv", index=False)
    print("✓ 绩效指标已保存至: ./performance_metrics.csv")

    # 5. 保存权重历史
    if 'weights_history' in backtest_results:
        weights_history = pd.DataFrame(backtest_results['weights_history'])
        weights_history.to_csv("./weights_history.csv", index=False)
        print("✓ 权重历史已保存至: ./weights_history.csv")

    # 6. 显示关键结果
    print("\n" + "=" * 60)
    print("关键绩效指标")
    print("=" * 60)

    metrics = backtest_results['metrics']
    print(f"最终投资组合价值: ${metrics.get('Final Portfolio Value', 0):,.2f}")
    print(f"初始资本: ${backtester.portfolio_manager.initial_capital:,.2f}")
    print(f"总收益率: {metrics.get('Total Return (%)', 0):.2f}%")
    print(f"年化收益率: {metrics.get('Annual Return (%)', 0):.2f}%")
    print(f"年化波动率: {metrics.get('Annual Volatility (%)', 0):.2f}%")
    print(f"夏普比率: {metrics.get('Sharpe Ratio', 0):.4f}")
    print(f"索提诺比率: {metrics.get('Sortino Ratio', 0):.4f}")
    print(f"最大回撤: {metrics.get('Max Drawdown (%)', 0):.2f}%")
    print(f"预测方向准确率: {metrics.get('Direction Accuracy', 0):.4f}")
```

```python
    print(f"交易次数: {metrics.get('Number of Trades', 0)}")

    # 7. 生成可视化图表
    generate_performance_charts(backtest_results)

def analyze_feature_stability(feature_importance_df):
    """分析特征重要性稳定性"""
    print("\n特征重要性稳定性分析:")

    # 按特征分组, 计算重要性的均值和标准差
    feature_stability = feature_importance_df.groupby('feature')['importance'].agg(['mean', 'std', 'count']).reset_index()
    feature_stability['cv'] = feature_stability['std'] / feature_stability['mean']  # 变异系数

    # 选择最重要的特征进行分析
    top_features = feature_stability.nlargest(20, 'mean')

    print("Top 20 特征稳定性:")
    for _, row in top_features.iterrows():
        stability = "高" if row['cv'] < 0.5 else "中" if row['cv'] < 1.0 else "低"
        print(f"  {row['feature']}: 均值={row['mean']:.4f}, 稳定性={stability}")

    # 保存特征稳定性分析
    feature_stability.to_csv("./feature_stability_analysis.csv", index=False)
    print("✓ 特征稳定性分析已保存至: ./feature_stability_analysis.csv")

def generate_performance_charts(backtest_results):
    """Generate performance charts"""
    try:
        import matplotlib.pyplot as plt
        import seaborn as sns
        import numpy as np
        import pandas as pd

        # Set style and parameters
        plt.rcParams['font.sans-serif'] = ['Arial', 'DejaVu Sans', 'Helvetica']
        plt.rcParams['axes.unicode_minus'] = False
        sns.set_style("whitegrid")


        # 1. Portfolio Value Curve
        plt.figure(figsize=(12, 6))
        plt.plot(backtest_results['portfolio_dates'], backtest_results['portfolio_values'],
                linewidth=2, color='#2E86AB')
        plt.title('Portfolio Value Over Time', fontsize=14, fontweight='bold')
        plt.xlabel('Date')
        plt.ylabel('Portfolio Value ($)')
        plt.xticks(rotation=45)
        plt.grid(True, alpha=0.3)

        # Format y-axis for better readability
        max_val = max(backtest_results['portfolio_values'])
        if max_val > 1e6:
            plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda x, p: f'${x/1e6:.1f}M'))
        else:
            plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda x, p: f'${x:,.0f}'))

        plt.tight_layout()
        plt.savefig('./portfolio_value_curve.png', dpi=300, bbox_inches='tight')
        plt.close()

        # 2. Prediction vs Actual Scatter Plot
        if 'predictions' in backtest_results and 'actual_returns' in backtest_results:
            predictions = backtest_results['predictions']
            actual_returns = backtest_results['actual_returns']

            if len(predictions) > 0 and len(actual_returns) > 0:
                min_len = min(len(predictions), len(actual_returns))
                plt.figure(figsize=(10, 6))

                # Create scatter plot with some styling
                plt.scatter(actual_returns[:min_len], predictions[:min_len],
                           alpha=0.6, s=30, color='#A23B72')

                # Add perfect prediction line
                min_val = min(min(actual_returns), min(predictions))
                max_val = max(max(actual_returns), max(predictions))
                plt.plot([min_val, max_val], [min_val, max_val], 'r--',
                        linewidth=2, alpha=0.8, label='Perfect Prediction')

                plt.xlabel('Actual Returns')
                plt.ylabel('Predicted Returns')
                plt.title('Predicted vs Actual Returns', fontsize=14, fontweight='bold')
                plt.legend()
                plt.grid(True, alpha=0.3)
                plt.tight_layout()
                plt.savefig('./prediction_vs_actual.png', dpi=300, bbox_inches='tight')
                plt.close()

        # 3. Daily Returns Chart
        portfolio_values = backtest_results['portfolio_values']
        if len(portfolio_values) > 1:
            daily_returns = []
            for i in range(1, len(portfolio_values)):
```

```python
                daily_return = (portfolio_values[i] - portfolio_values[i-1]) / portfolio_values[i-1]
                daily_returns.append(daily_return)

            plt.figure(figsize=(12, 6))
            plt.plot(backtest_results['portfolio_dates'][1:], daily_returns,
                    linewidth=1, color='#F18F01', alpha=0.8)
            plt.title('Daily Returns', fontsize=14, fontweight='bold')
            plt.xlabel('Date')
            plt.ylabel('Daily Return')
            plt.xticks(rotation=45)
            plt.grid(True, alpha=0.3)
            plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda x, p: f'{x:.2%}'))
            plt.tight_layout()
            plt.savefig('./daily_returns.png', dpi=300, bbox_inches='tight')
            plt.close()

        # 4. Drawdown Chart
        if len(portfolio_values) > 0:
            # Calculate drawdown
            portfolio_series = pd.Series(portfolio_values)
            rolling_max = portfolio_series.expanding().max()
            drawdown = (portfolio_series - rolling_max) / rolling_max

            plt.figure(figsize=(12, 6))
            plt.fill_between(backtest_results['portfolio_dates'], drawdown * 100, 0,
                           alpha=0.3, color='#C73E1D', label='Drawdown')
            plt.plot(backtest_results['portfolio_dates'], drawdown * 100,
                   color='#C73E1D', linewidth=1, alpha=0.8)
            plt.title('Portfolio Drawdown Over Time', fontsize=14, fontweight='bold')
            plt.xlabel('Date')
            plt.ylabel('Drawdown (%)')
            plt.xticks(rotation=45)
            plt.legend()
            plt.grid(True, alpha=0.3)
            plt.tight_layout()
            plt.savefig('./portfolio_drawdown.png', dpi=300, bbox_inches='tight')
            plt.close()

        # 5. Cumulative Returns Chart
        if len(portfolio_values) > 1:
            initial_value = portfolio_values[0]
            cumulative_returns = [(value - initial_value) / initial_value for value in portfolio_values]

            plt.figure(figsize=(12, 6))
            plt.plot(backtest_results['portfolio_dates'], cumulative_returns,
                    linewidth=2, color='#2E86AB')
            plt.title('Cumulative Returns', fontsize=14, fontweight='bold')
            plt.xlabel('Date')
            plt.ylabel('Cumulative Return')
            plt.xticks(rotation=45)
            plt.grid(True, alpha=0.3)
            plt.gca().yaxis.set_major_formatter(plt.FuncFormatter(lambda x, p: f'{x:.2%}'))
            plt.tight_layout()
            plt.savefig('./cumulative_returns.png', dpi=300, bbox_inches='tight')
            plt.close()

        print("Performance charts generated successfully!")
        print("portfolio_value_curve.png - Portfolio value over time")
        print("prediction_vs_actual.png - Prediction accuracy")
        print("daily_returns.png - Daily returns")
        print("portfolio_drawdown.png - Drawdown analysis")
        print("cumulative_returns.png - Cumulative returns")

    except ImportError:
        print("Warning: matplotlib or seaborn not installed, cannot generate charts")
        print("Please run: pip install matplotlib seaborn")
    except Exception as e:
        print(f"Error generating charts: {e}")

# 运行模型训练
if __name__ == "__main__":
    # 测试配置 - 简洁输出
    OutputConfig.VERBOSE = False
    OutputConfig.SHOW_PROGRESS = True
    OutputConfig.SHOW_FEATURE_DETAILS = False
    OutputConfig.SHOW_TRAINING_DETAILS = False

    main_model_training()
```

```
================================================================================
开始在线学习模型训练和回测流程
================================================================================

1. 加载数据...
数据维度: 2659 × 28001
时间范围: 2015-01-02 至 2025-07-30
目标变量: 2659 个有效样本

2. 模型训练与回测...
开始在线学习树模型策略

1. 初始化在线学习模型...
2. 初始化投资组合管理器...
3. 初始化策略回测器...
4. 运行回测...
回测期间: 2018-03-05 00:00:00 到 2025-07-30 00:00:00
回测数据量: 1862 个交易日
进行初始模型训练...
2018-03-05: 训练完成 - 756 样本, 23065 特征
============================================================
开始策略回测
============================================================
回测期间: 2018-03-05 至 2025-07-30
总交易日: 1862
进度: 1/1862 (0.1%)
动态特征优化: 从 23065 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
警告: 连续19次亏损
警告: 连续20次亏损
警告: 连续21次亏损
警告: 连续22次亏损
警告: 连续23次亏损
警告: 连续24次亏损
警告: 连续25次亏损
警告: 连续26次亏损
警告: 连续27次亏损
警告: 连续28次亏损
警告: 连续29次亏损
警告: 连续30次亏损
警告: 连续31次亏损
警告: 连续32次亏损
警告: 连续33次亏损
警告: 连续34次亏损
警告: 连续35次亏损
警告: 连续36次亏损
警告: 连续37次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续38次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 94/1862 (5.0%)
警告: 连续5次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
进度: 187/1862 (10.0%)
```

```
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 280/1862 (15.0%)
警告: 连续5次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 373/1862 (20.0%)
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 466/1862 (25.0%)
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 559/1862 (30.0%)
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
警告: 连续19次亏损
警告: 连续20次亏损
警告: 连续21次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 652/1862 (35.0%)
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
进度: 745/1862 (40.0%)
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
警告: 连续19次亏损
警告: 连续20次亏损
警告: 连续21次亏损
警告: 连续22次亏损
```

```
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
警告: 连续19次亏损
警告: 连续20次亏损
警告: 连续21次亏损
警告: 连续22次亏损
警告: 连续23次亏损
警告: 连续24次亏损
警告: 连续25次亏损
警告: 连续26次亏损
警告: 连续27次亏损
警告: 连续28次亏损
警告: 连续29次亏损
警告: 连续30次亏损
警告: 连续31次亏损
警告: 连续32次亏损
警告: 连续33次亏损
警告: 连续34次亏损
警告: 连续35次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续36次亏损
警告: 连续37次亏损
警告: 连续38次亏损
警告: 连续39次亏损
警告: 连续40次亏损
警告: 连续41次亏损
警告: 连续42次亏损
警告: 连续43次亏损
警告: 连续44次亏损
警告: 连续45次亏损
警告: 连续46次亏损
警告: 连续47次亏损
警告: 连续48次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
进度: 838/1862 (45.0%)
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续14次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 931/1862 (50.0%)
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1024/1862 (55.0%)
警告: 连续5次亏损
警告: 连续6次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
```

```
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1117/1862 (60.0%)
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1210/1862 (65.0%)
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
警告: 连续19次亏损
警告: 连续20次亏损
警告: 连续21次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
进度: 1303/1862 (70.0%)
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1396/1862 (75.0%)
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1489/1862 (80.0%)
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
进度: 1582/1862 (85.0%)
```

```
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1675/1862 (90.0%)
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
警告: 连续18次亏损
警告: 连续19次亏损
警告: 连续20次亏损
警告: 连续21次亏损
警告: 连续22次亏损
警告: 连续23次亏损
警告: 连续24次亏损
警告: 连续25次亏损
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1768/1862 (95.0%)
警告: 连续5次亏损
警告: 连续6次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
警告: 连续5次亏损
警告: 连续6次亏损
警告: 连续7次亏损
警告: 连续8次亏损
警告: 连续9次亏损
警告: 连续10次亏损
警告: 连续11次亏损
警告: 连续12次亏损
警告: 连续13次亏损
警告: 连续14次亏损
警告: 连续15次亏损
警告: 连续16次亏损
警告: 连续17次亏损
动态特征优化: 从 100 个特征中选择 100 个特征
进度: 1861/1862 (99.9%)
回测完成: 1862 次预测, 1862 次交易

5. 生成回测报告...

============================================================
回测结果报告
============================================================

核心绩效指标:
----------------------------------------

收益表现:
   总收益率: 188.89%
   年化收益率: 14.40%
   年化波动率: 2.65%
   夏普比率: 5.4342

风险控制:
   最大回撤: -8.83%
   索提诺比率: 8.4057

预测质量:
   预测相关性: 0.3817
   方向准确率: 0.6402

交易统计:
   交易次数: 1862
   交易胜率: 56.23%

资金信息:
   初始资本: $1,000,000
   最终价值: $2,888,862
   绝对收益: $1,888,862

3. 结果保存...
预测历史已保存至: ./prediction_history.csv
特征重要性历史已保存至: ./feature_importance_history.csv

特征重要性稳定性分析:
Top 20 特征稳定性:
  ret_21d_2633083_lag_21_roll_mean_21: 均值=0.0424, 稳定性=中
  ret_21d_10848661: 均值=0.0371, 稳定性=中
  ret_21d_34058542: 均值=0.0258, 稳定性=低
  ret_21d_47440465: 均值=0.0253, 稳定性=低
```

```
  ret_21d_2659551_lag_21_roll_mean_63: 均值=0.0233, 稳定性=中
  ret_21d_2661992_lag_1_roll_mean_21: 均值=0.0196, 稳定性=中
  ret_21d_2650430_roll_mean_5: 均值=0.0190, 稳定性=低
  ret_21d_248647424_lag_5_roll_mean_5: 均值=0.0189, 稳定性=低
  ret_21d_2656368: 均值=0.0180, 稳定性=中
  ret_21d_47440465_lag_1: 均值=0.0171, 稳定性=低
  financials_4256_ytd_557021597_roll_std_63: 均值=0.0168, 稳定性=低
  ret_21d_2591273: 均值=0.0167, 稳定性=低
  ret_21d_2659551_lag_5_roll_mean_63: 均值=0.0163, 稳定性=低
  ret_21d_2656368_roll_mean_5: 均值=0.0153, 稳定性=中
  financials_43898_ytd_2649739_lag_21_roll_std_63: 均值=0.0150, 稳定性=中
  financials_4430_ytd_2619516_lag_5_roll_mean_63: 均值=0.0150, 稳定性=中
  ret_21d_104552804_lag_21_roll_mean_63: 均值=0.0149, 稳定性=中
  ret_21d_2623683_lag_5_roll_mean_63: 均值=0.0149, 稳定性=中
  ret_21d_39458342_lag_5_roll_mean_63: 均值=0.0148, 稳定性=低
  ret_21d_2649166_lag_21_roll_mean_63: 均值=0.0146, 稳定性=中
✓ 特征稳定性分析已保存至: ./feature_stability_analysis.csv
✓ 投资组合结果已保存至: ./portfolio_results.csv
✓ 绩效指标已保存至: ./performance_metrics.csv
✓ 权重历史已保存至: ./weights_history.csv

============================================================
关键绩效指标
============================================================
最终投资组合价值: $2,888,861.76
初始资本: $1,000,000.00
总收益率: 188.89%
年化收益率: 14.40%
年化波动率: 2.65%
夏普比率: 5.4342
索提诺比率: 8.4057
最大回撤: -8.83%
预测方向准确率: 0.6402
交易次数: 1862
Performance charts generated successfully!
portfolio_value_curve.png - Portfolio value over time
prediction_vs_actual.png - Prediction accuracy
daily_returns.png - Daily returns
portfolio_drawdown.png - Drawdown analysis
cumulative_returns.png - Cumulative returns

============================================================
流程完成摘要
============================================================
最终绩效:
总收益率: 188.89%
年化收益率: 14.40%
夏普比率: 5.4342
最大回撤: -8.83%
预测准确率: 0.6402

交易统计:
交易次数: 1862
交易胜率: 56.23%

资金变化:
初始: $1,000,000
最终: $2,888,862
收益: $1,888,862

================================================================================
所有流程完成!
================================================================================
```