🧪 7.5 策略测试

> ***核心摘要：***
>
> *本节系统介绍TRQuant系统的策略测试系统，包括单元测试、集成测试和回测验证。通过理解策略测试的核心技术，帮助开发者掌握如何验证策略的正确性和性能，确保策略质量。*

策略测试是策略开发的重要环节，通过单元测试、集成测试和回测验证确保策略的正确性和性能。

📋 章节概览

🔬 *7.5.1 单元测试*

函数测试、逻辑测试、边界测试、异常测试

🔗 *7.5.2 集成测试*

模块集成测试、接口测试、数据流测试

📊 *7.5.3 回测验证*

回测框架、回测配置、回测执行、结果验证

🎯 学习目标

通过本节学习，您将能够：

- **编写单元测试**：掌握函数测试、逻辑测试和边界测试方法
- **进行集成测试**：理解模块集成测试和接口测试
- **执行回测验证**：掌握回测框架的使用和结果验证

📚 核心概念

模块定位

- **工作流位置**：步骤6 - 🛠️ 策略生成（策略规范化之后）
- **核心职责**：单元测试、集成测试、回测验证
- **服务对象**：策略优化、策略部署

🔬 7.5.1 单元测试

单元测试针对策略中的单个函数或逻辑单元进行测试，确保每个函数和逻辑单元的正确性。

测试框架

```
import pytest
import unittest
from unittest.mock import Mock, patch, MagicMock
import pandas as pd
import numpy as np

class StrategyUnitTest:
    """策略单元测试"""

    def test_initialize_function(self):
        """测试初始化函数"""
        # 创建模拟context
        context = Mock()
        context.portfolio = Mock()
        context.portfolio.total_value = 1000000
        context.portfolio.available_cash = 1000000
        context.portfolio.positions = {}
```

```
        # 执行初始化
        initialize(context)

        # 验证参数设置
        assert hasattr(context, 'max_position')
        assert context.max_position == 0.1
        assert hasattr(context, 'stop_loss')
        assert hasattr(context, 'take_profit')
        assert hasattr(context, 'universe')

    def test_rebalance_logic(self):
        """测试调仓逻辑"""
        context = Mock()
        context.universe = ['000001.XSHE', '000002.XSHE', '600000.XSHG']
        context.max_stocks = 10
        context.portfolio = Mock()
        context.portfolio.positions = {}
        context.portfolio.available_cash = 1000000
        context.portfolio.total_value = 1000000
        context.current_dt = pd.Timestamp('2024-01-01')

        # 模拟get_price返回数据
        with patch('get_price') as mock_get_price:
            mock_get_price.return_value = pd.DataFrame({
                'close': [10.0, 20.0, 30.0]
            })

            # 执行调仓
            rebalance(context)

            # 验证调仓结果
            assert len(context.portfolio.positions) <= context.max_stocks

    def test_risk_control_stop_loss(self):
        """测试止损逻辑"""
        context = Mock()
        context.stop_loss = 0.08
        context.portfolio = Mock()

        # 创建持仓
        position = Mock()
        position.avg_cost = 10.0
        position.total_amount = 1000
        context.portfolio.positions = {'000001.XSHE': position}

        # 模拟当前价格（触发止损）
        with patch('get_current_data') as mock_get_current:
            mock_get_current.return_value = {
                '000001.XSHE': Mock(last_price=9.0)  # 亏损10%
```

```
            }

            # 执行风控
            risk_control(context)

            # 验证止损触发
            # 应该卖出股票
            pass

    def test_factor_calculation(self):
        """测试因子计算函数"""
        stocks = ['000001.XSHE', '000002.XSHE']
        date = '2024-01-01'

        # 测试因子计算
        factor_values = calculate_factor(stocks, date)

        # 验证结果
        assert isinstance(factor_values, pd.DataFrame)
        assert len(factor_values) == len(stocks)
        assert not factor_values.empty
```

边界测试

```
class BoundaryTest:
    """边界测试"""

    def test_empty_universe(self):
        """测试空股票池"""
        context = Mock()
        context.universe = []
        context.max_stocks = 10

        # 应该不会报错
        rebalance(context)
        assert len(context.portfolio.positions) == 0

    def test_single_stock(self):
        """测试单只股票"""
        context = Mock()
        context.universe = ['000001.XSHE']
        context.max_stocks = 10

        rebalance(context)
        assert len(context.portfolio.positions) <= 1

    def test_extreme_parameters(self):
        """测试极端参数"""
        context = Mock()
        context.max_position = 1.0  # 100%仓位
```

```
        context.stop_loss = 0.01    # 1%止损
        context.take_profit = 1.0   # 100%止盈

        # 应该能正常处理
        initialize(context)
        assert context.max_position == 1.0
```

异常测试

```
class ExceptionTest:
    """异常测试"""

    def test_data_fetch_failure(self):
        """测试数据获取失败"""
        context = Mock()
        context.universe = ['000001.XSHE']

        # 模拟数据获取失败
        with patch('get_price', side_effect=Exception("数据获取失败")):
            # 应该优雅处理异常
            rebalance(context)
            # 不应该崩溃

    def test_invalid_stock_code(self):
        """测试无效股票代码"""
        context = Mock()
        context.universe = ['INVALID_CODE']

        # 应该跳过无效代码
        rebalance(context)
        assert 'INVALID_CODE' not in context.portfolio.positions
```

🔗 7.5.2 集成测试

集成测试验证策略各模块之间的集成是否正确，包括模块集成测试、接口测试和数据流测试。

完整工作流测试

```
class StrategyIntegrationTest:
    """策略集成测试"""

    def test_strategy_workflow(self):
        """测试策略完整工作流"""
        # 1. 初始化
        context = create_test_context()
        initialize(context)

        # 验证初始化结果
        assert hasattr(context, 'universe')
        assert len(context.universe) > 0
```

```
        # 2. 模拟多个交易日
        test_dates = pd.date_range('2024-01-01', '2024-01-10', freq='B')

        for date in test_dates:
            context.current_dt = date

            # 盘前处理
            before_market_open(context)

            # 开盘处理
            market_open(context)

            # 盘后处理
            after_market_close(context)

        # 3. 验证最终状态
        assert context.portfolio.total_value > 0
        assert len(context.portfolio.positions) <= context.max_stocks

    def test_module_integration(self):
        """测试模块集成"""
        context = create_test_context()

        # 测试选股模块与调仓模块的集成
        selected_stocks = select_stocks(context)
        rebalance(context, selected_stocks)

        # 验证集成结果
        assert all(
            stock in context.portfolio.positions
            for stock in selected_stocks[:context.max_stocks]
        )

    def test_data_flow(self):
        """测试数据流"""
        context = create_test_context()

        # 1. 数据获取
        market_data = get_market_data(context.universe)
        assert market_data is not None

        # 2. 因子计算
        factor_scores = calculate_factor_scores(market_data)
        assert factor_scores is not None

        # 3. 选股
        selected = select_stocks_by_scores(factor_scores)
        assert len(selected) > 0
```

```
        # 4. 调仓
        rebalance(context, selected)

        # 验证数据流完整性
        assert len(context.portfolio.positions) > 0
```

📊 7.5.3 回测验证

回测验证通过历史数据验证策略性能，确保策略在历史数据上的表现符合预期。

回测配置

```
from core.backtest import BacktestEngine

def test_backtest_validation():
    """回测验证测试"""
    # 配置回测参数
    backtest_config = {
        'start_date': '2020-01-01',
        'end_date': '2023-12-31',
        'initial_capital': 1000000,
        'benchmark': '000300.XSHG',
        'commission_rate': 0.0003,
        'slippage': 0.002
    }

    # 执行回测
    engine = BacktestEngine(backtest_config)
    backtest_result = engine.run(strategy_code)

    # 验证回测结果
    assert backtest_result['sharpe_ratio'] > 1.0
    assert backtest_result['max_drawdown'] < 0.20
    assert backtest_result['total_return'] > 0
    assert backtest_result['win_rate'] > 0.5
```

回测结果验证

```
class BacktestResultValidator:
    """回测结果验证器"""

    def validate_backtest_result(
        self,
        backtest_result: Dict[str, Any]
    ) -> Tuple[bool, List[str]]:
        """
        验证回测结果

        Args:
            backtest_result: 回测结果
```

```
        Returns:
            Tuple[bool, List[str]]: (是否有效, 错误列表)
        """
        errors = []

        # 检查必需指标
        required_metrics = [
            'total_return', 'sharpe_ratio', 'max_drawdown',
            'win_rate', 'total_trades'
        ]

        for metric in required_metrics:
            if metric not in backtest_result:
                errors.append(f"缺少必需指标: {metric}")

        # 验证指标合理性
        if 'sharpe_ratio' in backtest_result:
            sharpe = backtest_result['sharpe_ratio']
            if sharpe < -5 or sharpe > 10:
                errors.append(f"夏普比率异常: {sharpe}")

        if 'max_drawdown' in backtest_result:
            max_dd = backtest_result['max_drawdown']
            if max_dd < 0 or max_dd > 1:
                errors.append(f"最大回撤异常: {max_dd}")

        return len(errors) == 0, errors
```

🔗 相关章节

- **7.4 策略规范化**：了解策略规范化，测试基于规范化后的策略
- **第8章：回测验证**：了解回测验证的详细内容

💡 关键要点

1. **单元测试**：确保每个函数和逻辑单元的正确性
2. **集成测试**：验证模块之间的集成是否正确
3. **回测验证**：通过历史数据验证策略性能

🔮 总结与展望

本节回顾

本节系统介绍了策略测试系统，包括单元测试、集成测试和回测验证。通过理解策略测试的核心技术，帮助开发者掌握如何验证策略的正确性和性能，确保策略质量。

下节预告

掌握了策略测试后，下一节将介绍策略部署，包括策略打包、策略上传和策略监控。通过理解策略部署的核心技术，帮助开发者掌握如何将策略部署到实盘交易平台。

继续学习：7.6 策略部署 →

> ***适用版本**: v1.0.0+*
> ***最后更新**: 2025-12-12*