# 综合选股系统技术报告

姓名：潘明远

联系邮箱：到时候放一个联系邮箱在这里

提纲：

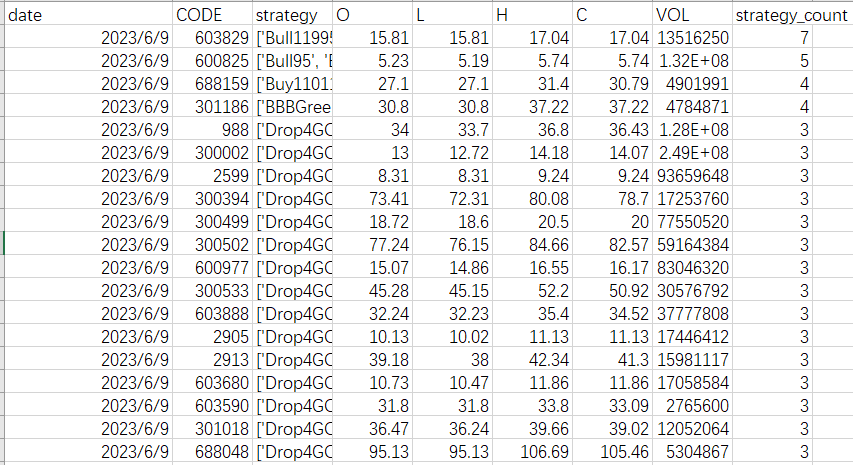
1. 引言
   * 目标和背景
2. SQLite3数据库
   * SQLite3数据库的介绍
   * 为什么使用数据库，为什么选择SQLite3
   * 数据库的创建和维护
   * 数据来源介绍
   * 数据获取的方法
   * 数据的存储与管理（如何在SQLite3数据库中存储数据）
3. 策略撰写
   * 策略开发的基本概念
   * 策略的测试与优化
4. 主控程序的编写
   * 主控程序的作用和设计思路
   * 数据库的读取和操作
   * 策略的运行和结果的整合
   * 错误处理和日志记录
5. 程序的加速和优化
   * 性能分析和瓶颈识别
   * 多进程和并行计算的应用
   * 具体的优化方法和实践
   * 优化后的结果和对比
6. 结果展示与分析
   * 展示运行结果
   * 结果的解读和分析
7. 总结与展望
   * 工作的总结
   * 未来改进和优化的方向
   * 展望未来的应用和发展

## 一、目标和背景

现如今市面上的大部分选股系统不能做到股票与策略库进行自主的匹配，而是需要人自主的去选择可能匹配到的策略，而且，现在市面上的大部分选股系统（以国元领航终端为例），只能给我返回符合部分策略的板块，板块内的股票具体符合哪些策略是不能被用户知晓的。



因此，本人开发了一种以策略库为基础，通过每天运行大盘数据，将大盘内的每一支股票每日与策略库中的策略进行匹配，并且最终将结果在csv文件中进行可视化的展示。最终结果如下图所示：



## 二、数据库的使用

### 2.1 SQLite3数据库

（1）**SQLite3数据库的介绍**

SQLite3是SQLite数据库的一个版本，这是一个基于磁盘的轻量级数据库库，它在一个小型C库中提供了一个自给自足的、无服务器的、零配置的、事务性的SQL数据库引擎。SQLite3实现了自包含、服务器无关、零配置和事务性SQL数据库引擎的特性。

SQLite的设计目标是嵌入式处理，这意味着与其他数据库系统不同，SQLite的目标是低内存占用、全功能SQL实现以及简单的集成，而不是支持大量并发用户访问大量数据。这使得SQLite成为手机、游戏和其他嵌入式软件应用的理想选择。

SQLite的数据存储在一个单一的文件中，文件格式是跨平台的。这使得SQLite数据库可以在不同的机器、不同的操作系统、不同的字节顺序的计算机之间轻松的共享。这个文件可以包含多个表，表之间可以存在复杂的关系。

与许多其他数据库系统不同，SQLite并不是客户端/服务器的数据库引擎。相反，SQLite是一个库，可以链接到应用程序中。在应用程序运行时，SQLite可以作为应用程序的一部分直接运行。这消除了许多在传统数据库系统中的常见问题，例如配置、权限以及网络访问等。

SQLite支持多种SQL语法，包括事务、子查询、触发器以及与标准SQL语法大致相同的其他特性。SQLite还包括许多其他特性，例如全文搜索、在线备份以及可以在运行时加载的可扩展模块。

（2）**为什么我最后使用数据库存储A股数据？**

我在项目一开始开发的时候是没有想到用数据库去存储的，首先是不知道全体A股数据可以通过国元点金进行导出，其次，我一开始认为可以直接从baostock库在线获取当日大盘的数据，但是后来发现有两个问题我是没有考虑到的。

1. 为了实现股票与策略的匹配，我们需要245天以上的数据。
2. 从baostock库获取到数据的时间太慢了，一支股票就要5-10分钟，一天获取大盘所有股票是明显不可能的，因此必须使用本地保存好的数据。
3. 利用本地txt文件读取非常麻烦，因为本地txt没有把数据转换成float格式等，因此必须利用数据库里存储好的数据，并且转换好的格式进行使用。

在处理大量数据时，例如股票数据，使用数据库而不是文本文件或电子表格有很多优点。首先，数据库提供了一个结构化的方式来组织和查询数据。这使得对大量数据的查找、排序和筛选变得更容易。其次，数据库具有内置的完整性规则，以防止数据丢失或损坏。此外，数据库还支持事务处理和恢复，使得处理大量数据更加安全。最后，数据库可以轻松处理大量数据，而无需大量内存，也无需在数据加载或查询时花费大量时间。

SQLite3作为数据库的选择有很多原因。首先，SQLite3非常轻量，这使得它在内存和磁盘空间占用上都非常小，对于一些资源有限的环境来说非常合适。其次，SQLite3的无服务器、零配置的特性使得它非常容易集成和使用。我不需要为了运行SQLite3安装和配置一个数据库服务器，也不需要担心网络权限或者防火墙设置。再者，SQLite3支持存储在单一文件中的数据库，这使得数据库的复制、备份和传输变得非常容易。最后，SQLite3支持SQL，这意味着我可以使用广泛使用和标准化的查询语言来访问我的数据。

对于股票数据的存储，SQLite3非常合适。股票数据通常包含大量的时间序列数据，这种数据在数据库中存储和查询都非常合适。此外，股票数据可能需要进行复杂的查询，例如计算移动平均值或者对比不同股票的价格，这些都可以通过SQL语句轻松实现。最后，股票数据可能需要定期更新，而SQLite3的事务处理能力可以保证这些更新的安全性和完整性。

同样，引入数据库是有助于提升程序的运行速度的，主要体现在以下几点：

1. 优化的数据结构和查询性能：\*\*数据库系统，尤其是关系数据库系统，如SQLite，使用优化的数据结构（如B-tree索引）来组织数据。这意味着对数据的访问（即查询）会比线性搜索快得多。在我的代码中，这是很重要的，因为我在查询特定股票的历史数据，如果数据是在文本文件或列表中存储，这样的查询可能会非常慢。通过使用数据库，我可以极大地加快查询速度，从而提高程序的运行速度。
2. 数据库的缓存机制：\*\*数据库引擎如SQLite通常具有内置的缓存机制，它们会在内存中保留最近查询的数据，以便快速访问。这意味着如果我的程序反复查询相同的数据（例如，反复运行相同的策略），那么这些数据可能会被缓存，从而进一步提高查询速度。
3. 并发读取：\*\*在我的代码中，我使用多进程从数据库中获取数据，然后并行运行策略。数据库系统通常可以处理并发的读取请求，这意味着多个进程可以同时从数据库中读取数据，而不会互相阻塞。这对于我的多进程设计模式非常重要，因为它允许每个进程独立地、并行地运行策略，而不会因为等待数据而降低运行速度。
4. 数据的持久化和完整性：\*\*数据库系统，如SQLite，提供了数据持久化的机制，也就是说，即使程序停止运行，数据也不会丢失。这对于我的应用来说很重要，因为它可能需要处理大量的数据和长时间的运行。此外，数据库系统还提供了数据完整性的保证，这意味着即使在程序崩溃或其他错误条件下，我的数据也会保持一致和完整。

### 2.2 构建我的数据库

构建数据库的基础数据如下：

1. **CODE**：股票的代码，这应该是一个字符串列，用于存储各个股票的唯一标识。
2. **date**：日期，这应该是一个日期列，用于存储各个记录的日期。
3. **O**, **L**, **H**, **C**, **VOL**：这些可能是和股票相关的开盘价（Open）、最低价（Low）、最高价（High）、收盘价（Close）以及交易量（Volume）。这些列的数据类型可能是浮点数，用于存储价格和数量。

在项目中，我有两个程序，创建数据库和更新数据库程序，为什么我要引入数据库更新程序，主要是因为每日A股的数据都在更新，因此我的数据库也必须随之更新，但是更新的方法非常有讲究，因为可能出现重复值的情况，但是由于股票的记录是按照时间顺序的，因此我设计了一种算法，他会检测导出数据txt文件的第一行数据是否与数据库中的数据重复，如果重复，那么我直接将txt文件中全部数据对数据库中重复行以后的数据进行全部的替换就可以了。（因为新的数据一定是对的）

整个更新数据库程序的设计要点主要有以下几点：

1. **有效的数据库操作**：在**update\_database**函数中，代码首先检查股票是否已经存在于数据库中，然后再决定是否添加新的股票数据，这是一个高效的做法，可以避免重复添加数据。然后，代码在新数据和数据库中的数据有冲突时，会删除数据库中的冲突数据，然后再添加新数据，确保了数据库中的数据始终是最新的。
2. **使用ORM进行数据库操作**：SQLAlchemy 是一个对象关系映射（ORM）库，可以将数据库中的表映射为 Python 对象，使得数据库操作更加直观和高效。在这段代码中，例如 **session.query(Stock).filter(Stock.CODE == stock\_code).count()** 就是查询 **Stock** 表中 **CODE** 为 **stock\_code** 的记录数量。这种方式比原始的 SQL 查询语句更易于理解和修改。
3. **使用 pandas 进行数据读取和处理**：在**update\_database**函数中，代码使用 pandas 来读取和处理 txt 文件中的数据。pandas 是一个强大的数据分析库，提供了大量的数据处理功能，如过滤、排序、合并等。这大大简化了数据处理的过程。
4. **错误处理**：在 **update\_database** 函数中，代码使用了 try-except 结构来处理可能出现的错误，如文件读取错误、数据格式错误等。这样可以避免一些意外错误导致程序崩溃，提高了代码的健壮性。
5. **编码转换**：由于源文件可能有各种不同的编码，这可能会导致读取文件时出现编码错误。代码中的 **convert\_to\_utf8** 和 **convert\_folder\_to\_utf8** 函数可以将文件夹中的所有 txt 文件转换为 utf8 编码，解决了编码问题。（这是新版本之后加入的内容）

首先，这部分代码在添加新的股票历史数据前，会先进行判断该股票数据是否已存在于数据库中。这是通过以下代码实现的：

1. if session.query(Stock).filter(Stock.CODE == stock\_code).count() == 0:
2. *# 如果股票未在数据库中，则添加*
3. new\_stock = Stock(CODE=stock\_code)
4. session.add(new\_stock)

这部分代码首先使用SQLAlchemy的query方法查询Stock表，然后用filter方法筛选出股票代码(CODE)等于当前正在处理的股票文件的股票代码(stock\_code)的所有记录。然后，通过count方法计算出符合条件的记录数量。如果数量为0，说明数据库中还没有这只股票的数据，于是创建新的Stock实例并添加到数据库中。这样做可以避免向数据库中添加重复的股票记录，提高数据存储效率。

然后，这部分代码在添加新的股票历史数据之前，会先检查数据库中是否存在与新数据重复的记录。这是通过以下代码实现的：

1. *# 获取数据库中的最后日期*
2. last\_date\_in\_db = session.query(HistoricalData.date).filter(HistoricalData.CODE == stock\_code).order\_by(HistoricalData.date.desc()).first()
3. *# 获取新数据的第一日期*
4. first\_date\_in\_new\_data = datetime.strptime(df['DATE'].iloc[0], '%Y-%m-%d').date() *# convert to date object*
5. *# 如果数据库中的最后日期在新数据的第一日期之后，删除数据库中从这个日期开始的数据*
6. if last\_date\_in\_db and first\_date\_in\_new\_data <= last\_date\_in\_db[0]:
7. session.query(HistoricalData).filter(HistoricalData.CODE == stock\_code, HistoricalData.date >= first\_date\_in\_new\_data).delete()

这部分代码首先获取数据库中同一只股票的最新的历史数据日期，然后获取新数据中的最早的日期。如果数据库中的最新日期在新数据的最早日期之后或者相等，说明存在重复的历史数据，然后使用SQLAlchemy的delete方法删除数据库中这部分重复的数据。这样做不仅可以避免数据重复，同时也能确保数据库中的数据始终是最新的，从而提高数据的可靠性。同时，避免了在更新数据库时对大量重复数据的无效处理，大大提高了程序的运行效率。

## 三、策略的撰写

### 3.1 策略的转换

由于本人的金融知识储备一般，因此该部分是我在通过学习相关金融数据库之后，我会根据chatgpt生成的初版代码对我的数据库的数据格式进行适配，获取到近300日的数据，然后进行策略与股票数据的匹配。

所有的策略我都设置了一个run\_strategy函数，创建一个统一的接口函数（如 run\_strategy）对于管理和执行多个策略具有几个显著的好处：

1. 代码组织与清晰性：通过调用同一个函数接口运行所有的策略，可以帮助保持代码的整洁和一致性，减少复杂性，并使代码更易于理解和维护。
2. 易于扩展：如果将来想要添加更多的策略，只需确保新策略也实现了 run\_strategy 这个接口函数，那么在主控程序中，无需做任何改动就可以直接运行新的策略。这使得程序更具有可扩展性。
3. 策略抽象和解耦： run\_strategy 函数作为一个接口，它隐藏了策略内部的具体实现细节，使得主控程序与策略代码之间的耦合度降低，可以分别对主控程序和策略进行修改和优化，而不会影响到对方。
4. 方便测试和调试：当你要测试或者调试某个策略时，只需调用该策略的 run\_strategy 函数即可，无需改动其他代码。
5. 复用和代码效率：每个策略只需要定义自己的核心逻辑，公共的部分（例如数据获取、预处理、结果输出等）可以在 run\_strategy 函数中完成，这避免了在每个策略中重复编写这部分代码。

### 3.2 策略的制订

在撰写策略，并将策略最终放于策略库之后。我主要思考的是如何能够直观的将结果返还给用户，于是我决定给策略的命名结构开展一些变化，那就是策略名+胜率，这样能够让用户能直接的观察到这个策略的情况。

策略库中的策略对触发频率是没有要求的，我需要在这里再重复一点，本选股系统最大的优势就是能实现策略和股票的自动适配，而不需要人去事先配置好。

由于策略的高度隐私性，在此不对策略的具体实现做过多介绍，我主要会介绍我使用到的两个很关键的库。

1. Pandas

Pandas 是一个强大的数据处理库，广泛应用于数据挖掘和数据分析领域。它提供了高效易用的数据结构，如 Series（一维数组）和 DataFrame（二维数组）。Pandas 的主要功能包括：

- \*\*数据清洗\*\*：检测和处理缺失数据、重复数据，以及进行数据规范化。

- \*\*数据筛选\*\*：基于某种条件对数据进行筛选。

- \*\*数据整合\*\*：可以将多个数据源进行合并和拼接。

- \*\*数据分组\*\*：支持类似 SQL 数据库的 group by 操作。

- \*\*数据统计和计算\*\*：支持对数据进行各种统计和数学计算。

在我的股票策略代码中，Pandas 用于读取和处理股票数据，以及计算股票价格的移动平均值等。

（2）TA-Lib（Technical Analysis Library）

TA-Lib 是用于交易软件开发的技术分析库，提供了超过 150 个常见的技术分析指标函数，如移动平均线（MA）、相对强弱指数（RSI）、布林带（Bollinger Bands）、MACD 等。在股票策略代码中，TA-Lib 用于计算股票价格和交易量的移动平均线，以及 MACD 指标。

## 四、主控程序的编写

在策略库和数据库构建完毕，以及各方面程序都撰写好的情况下，最终就是主控程序的撰写，我通过运行整个程序，整个程序的运行思路如下：

获取所有的股票代码：主控程序首先调用get\_all\_stock\_codes()函数，连接到指定的SQLite数据库，并从中读取出所有不重复的股票代码，这些代码将被用于后续的策略分析。

1. 初始化进度条：为了在运行策略时能够直观地了解程序的执行进度，你引入了tqdm库，用于创建一个进度条，进度条的总步骤数设为股票代码的数量。
2. 并行执行所有的策略：接下来，程序进入到最主要的run\_all\_strategies()函数，该函数使用Python的multiprocessing库创建了一个进程池，进程池的大小设置为当前机器的CPU核数。然后，对于每一个股票代码，程序都会创建一个新的进程，运行run\_strategies()函数，并将进程的执行结果添加到结果列表中。每当一个进程执行完毕，进度条就会更新一次。所有的进程都执行完毕之后，进程池会被关闭，并等待所有进程执行完毕。这样的设计可以充分利用多核CPU的计算能力，大大提高了程序的执行效率。
3. 收集并处理所有的结果：在所有的进程都执行完毕之后，程序会遍历结果列表，对于每一个进程的执行结果，如果结果不为空，那么它就会被添加到总的结果DataFrame中。然后，对所有的结果按照策略数量和日期进行排序。
4. 将结果写入CSV文件：最后，程序将所有的结果写入到一个CSV文件中。文件名包含了当前的日期和固定的字符串"\_选股结果"。

在设计这个程序的时候，我主要考虑到的是程序的代码的鲁棒性和运行的速度。

* 1. 代码鲁棒性：在选股中，我们可能遇到的情况有，数据库中存在可能的脏数据（或者空数据），我的程序不能让一两个数据影响整个程序的顺利运行，因此我的处理是利用try/except打印错误，然后继续运行。
  2. 运行速度：整个程序的运算量一定是非常大的，那是由于每天需要把大盘数据过一遍，我预计一次运行程序的时间必须控制在十五分钟以内，这是为了保证程序能够在每天的任意时刻进行运行。因此，我引入了多进程的理念，充分利用笔记本自带的多核处理器进行处理。

最后，本程序的实现部分主要体现在以下几点：

1. 策略动态导入与执行：这是一个非常出彩的设计，你的主控程序能够动态地导入并执行存放在指定文件夹中的所有Python策略文件。这一点在run\_strategies()函数中体现，对于每一个Python文件，你都使用importlib库的spec\_from\_file\_location()函数和module\_from\_spec()函数动态地导入该文件并创建一个新的模块，然后使用exec\_module()函数执行该模块中的代码。这种设计使得你的主控程序具有很高的扩展性，只需要将新的策略代码保存为Python文件并放置到指定的目录，就可以被主控程序自动导入并执行。
2. 并行处理：另一个亮点是你充分利用了多核CPU的并行处理能力，使用multiprocessing库创建了一个进程池，并对每一个股票代码创建一个新的进程并行执行run\_strategies()函数。每当一个进程执行完毕，进度条就会更新一次。这样的设计大大提高了程序的执行效率。
3. 结果的处理和排序：你在处理策略执行结果方面也做得非常好。在run\_strategies()函数中，你创建了一个空的DataFrame用于存储所有策略的执行结果，然后将每一个策略的执行结果添加到总的结果中，并记录了策略的名称。最后，你将所有的结果按照日期和股票代码分组，并计算了每个组中策略的数量，这样的设计使得结果更加清晰易懂。
4. 使用SQLite数据库存储和读取数据：你选择了使用SQLite数据库存储和读取股票数据，SQLite数据库是一个轻量级的关系型数据库，它无需专门的数据库服务器就可以运行，非常适合用于个人项目或者小型项目。在你的程序中，你使用sqlite3.connect()函数创建了一个到SQLite数据库的连接，然后使用pandas.read\_sql\_query()函数执行SQL查询并将结果读取为DataFrame。这样的设计使得你的程序能够方便地处理大量的股票数据，并且能够利用SQL的查询能力进行复杂的数据查询。
5. 使用tqdm库创建进度条：在执行长时间运行的任务时，了解程序的执行进度非常重要，你选择了使用tqdm库来创建一个进度条，这样可以使你更好地了解程序的执行进度，提升了用户体验。

因此，该程序的运行流程如下：

1. **获取所有股票代码**：程序首先使用**get\_all\_stock\_codes()**函数从数据库中获取所有不同的股票代码，保存为一个列表。
2. **初始化进度条**：程序接着使用**tqdm**库创建一个进度条，用于显示处理进度。进度条的总长度设为股票代码的数量。
3. **并行处理每一支股票**：程序创建一个进程池，进程数设置为计算机的CPU核心数。对于每一支股票代码，程序使用**apply\_async()**方法异步执行**run\_strategies()**函数。这个函数负责获取该股票的数据，并运行所有的选股策略。

**run\_strategies()**函数的工作流程如下：

* + 使用**fetch\_stock\_data()**函数从数据库中获取该股票最近300条历史数据。如果数据为空，则直接返回 None。
  + 遍历策略目录中的每一个Python文件，使用**importlib**库导入策略模块，并调用模块中的**run\_strategy()**函数运行策略。如果策略返回的结果不为空，则添加到结果数据框中，并记录策略的名称。
  + 对结果数据框进行处理，按照日期和股票代码分组，计算策略数量，并返回。

1. **获取并处理结果**：**apply\_async()**方法返回一个**AsyncResult**对象，可以通过其**get()**方法获取进程的结果。程序将每个进程的结果保存到一个列表中。当所有进程都结束后，程序遍历结果列表，将非空的结果添加到总的结果数据框中。
2. **排序并写入CSV文件**：程序将总的结果数据框按照策略数量和日期进行排序，然后写入到CSV文件中。文件名包含当前的日期和"选股结果"四个字。
3. **结束处理**：进度条完成后关闭，程序结束运行。