Оглавление

[Внутреннее устройство 2](#_Toc51237810)

[Основные типы 2](#_Toc51237811)

[Алгоритмы работы 2](#_Toc51237812)

[Вывод лога 2](#_Toc51237813)

[FileSM 3](#_Toc51237814)

[Разбор outputPattern 3](#_Toc51237815)

[Пакетная обработка 4](#_Toc51237816)

[Фиксация заполненности очереди 4](#_Toc51237817)

[Самодиагностика 4](#_Toc51237818)

[Эксплуатация 5](#_Toc51237819)

[Подключение 5](#_Toc51237820)

[Создать кастомный loglevel 5](#_Toc51237821)

[Изменить цвета logLevel-ов 5](#_Toc51237822)

[Исправление коллизии при добавлении записи в БД 5](#_Toc51237823)

[Конфиг-файл 6](#_Toc51237824)

[path 6](#_Toc51237825)

[outputPattern 6](#_Toc51237826)

[time 6](#_Toc51237827)

[usingStores 6](#_Toc51237828)

[insideLoggingStore 7](#_Toc51237829)

[Добавление store-ов 7](#_Toc51237830)

[Изменить параметры пакетной обработки 7](#_Toc51237831)

[Изменить название хранилища в конфиге 7](#_Toc51237832)

[Изменить символы начала-окончания селекторов 7](#_Toc51237833)

[Добавить селекторы и выводимую информацию 7](#_Toc51237834)

[Изменить логируемую самодигностическую инфу 8](#_Toc51237835)

[Изменить предупредительный порог занятости очереди обработки 8](#_Toc51237836)

[Добавить новые параметры в конфиг 8](#_Toc51237837)

[Изменить корневую папку логов 8](#_Toc51237838)

Stores:

-file

-console

-MongoDb

Не работает с интерфейсом логгеров ASP ILogger, поскольку:

* нужно легко добавлять самый разнообразный функционал,
* логгер должен работать не только с проектами ASP.

Создан как открытый для редактирования под себя проект либы, а не закрытая либа (чёрный ящик).

# Внутреннее устройство

## Основные типы

* StruLogProvider – класс инициализации либы,
* Logger – класс, определяющий каждый логгер, управляет выводом в store-ы,
* LoggersFactory – класс, управляющий созданием логгеров,
* LoggersExtensions – методы-расширения для удобного вывода логов вместо использования тяжеловесного метода Log() в классе Logger.
* ConfigFileProvider (static class для парса),
* StoreManager (abstract) – общий функционал для всех SM-ов:
  + StringStoreManager (abstract) – функционал для работы со строковыми хранилищами, где данные хранятся/выводятся строками
    - ConsoleSM (Singleton)
    - FileSM (Singleton)
  + DbStoreManager (abstract) – функционал для БД-хранилищ - характеризуются тем, что используют модельные типы для хранения (SQL или NoSQL – не имеет значения):
    - MongoSM (Singleton)
* IBatchProcessingCompatible – интерфейс, который имплементируют классы StoreManager-ы, реализующие пакетную обработку. Если новый StoreManager с пакетной обработкой не будет имплементировать интерфейс, он окажется несовместимым с уже имеющимся кодом.
* LogsBatchProccessor – класс, предоставляющий обобщённый функционал по пакетной обработке логов – содержит методы логирования разных типов, все выполняют одну и тоже конечную цель, но по-разному (кроме непосредственно соединения с хранилищем и записи в него лог-записи, там есть логика задержек между попытками что-то сделать, отслеживания занятости очереди, логирования событий), для пакетного вывода логов нужно использовать какой-то конкретный метод из него для Storemanager-а. Данный класс можно и не использовать, если требуется особая реализация обработки, однако вряд ли она на столько специфична, что её не нужно обобщать – лучше определить её в этом классе как отдельный метод.

## Алгоритмы работы

### Вывод лога

1. Подключение:
   1. подключили либу,
   2. задали конфиг на стороне использующего приложения,
   3. вызвали инициализацию в Program.cs с помощью InitStruLog(). Распарсился конфиг.
   4. добавили логгеры как static readonly в нужные классы, привязанные к типу логируемого класса,
2. Во время инициализации экземпляра логгера (static readonly) идёт запрос в concurrent-словарь (хранится в Logger), который если не имеет логгер с соотв. ключом (название типа – ключ словаря), создаёт его, вызывая приватный конструктор класса. 1 logger on 1 class
3. ConcurrentDictionary with loggers – иногда, особенно в контроллерах ASP, логгеры могут быть запрошены примерно в одно время и если будет лишь один экземпляр логгера, мы будем дольше его ожидать, использование Concurency немного ускоряет процесс.
4. Вызвали метод логирования:
   1. logger.{logLevelName}(params\_i) – метод расширения
   2. logger.{logLevelName}(params\_i+1) – метод расширения
   3. logger.Log(LogLevel, message, object, Exception)
5. Все методы расширения работают с методом 4c, который синхронизирован lock/Monitor-ом (thread safe). Он находится в классе Logger.
   1. Почему не спиновая блокировка? Была практически изучена производительность Визуализатором параллелизма и при использовании спин-блокировки производительность хуже.
6. StoreManager-ы не зависят от logger-ов и существуют в единичных экземплярах (Singleton-ы, чтобы использовать плюсы ООП и статических классов). В Log() к ним имеется доступ. Включенным SMам (в конфиге это прописано в поле usingStores) передаются нужные аргументы и запускается обработка (вызов общего для StoreManager-ов метода TryLog()), для класса Logger не имеет значения пакетная обработка или синхронная основным потоком.
7. Если Пакетная обработка:
   1. TryLog() добавляет данные в очередь,
   2. заранее должен был быть запущен RunBatchProcessing, который запускает фоновый поток для StoreManager-а, имплементирующего IBatchProcessingCompatible. Запускается таким образом метод вывода логов из очереди, определённый в конкретном StoreManager-е, как правило это метод Log()
   3. В методе Log() SM-а нужно организовать пакетную обработку самостоятельно (разбор очереди), либо воспользоваться обобщённым функционалом класса LogsBatchProccessor.
   4. Например, в LogsBatchProccessor имеется метод Log\_Type1Async(Func<Task> ConnectTo\_Func, Func<TLogEntry, LogData, Task> WriteLogEntryTo\_Func), он требует логику подключения к хранилищу (файлу, базе…) и логику записи в неё. Он сообщает о загруженности очереди, логирует события и делает задержки между попытками совершить действия с хранилищем, зацикливает последний элемент массива времён задержек (они указаны в миллисекундах), когда до него добирается – т.е. задержка между попытками становится одной и тоже продолжительности. Чтобы воспользоваться этим вспомогательным методом, в методе Log() нужно создать объект LogsBatchProccessor-а, определить нужные анонимные методы и вызвать его в LogsBatchProccessor-е.
8. Если синхронная обработка (основной поток выводит):
   1. TryLog() может просто включать в себя прямой вызов метода Log(args),
   2. такой подход актуален, если основной поток почти совсем не задержится для вывода, но даже консоль может задержать основной поток с выводом.

#### Формирование лог-записи

outputPattern - это параметр из конфига, определяющий содержание записи, он задействуется при каждом выводе лога, но не в сыром виде: при парсинге конфига его содержимое превращается в последовательность анонимных методов, т.е. сразу определяется что выводится и в каком порядке, дабы постоянно не тратить время на побитовый проход по строке паттерна.

### FileSM

* Удерживает дескриптор файла все время логирования
* Пересоздаёт лог файлы с определённой периодичностью
* Сохраняет в файл дату создания, поэтому после рестарта логгера корректно заменит файл. Если сохранение или импорт из файла с датой пройдут с ошибкой, после первой попытки больше попыток не будет, а программа продолжит корректно работать.
* Пересоздание возможно не чаще 1 дня.
* Если создаваемый файл совпадает по названию с уже имеющимся, новый не создаётся, информация добавляется в уже имеющийся файл.
* Включена функция AutoFlush, чтобы в файле логов всегда были самые последние экспортированные логи (буфер не используется).

### Разбор outputPattern

Работает только при парсинге конфига.

Рассматривались 2 варианта:

1. побайтовый проход и использование как символа открытия {, так и символа закрытия } [используется, ибо быстрее]
2. использование лишь символа открытия % и поиск нужных селекторов регулярными выражениями: в лучшем случае получаем побайтовый проход, но уже с учетом избранных нами вероятных символов, которые должны присутствовать в селекторах: начинаем проход по паттерну рег. выражением селектора, получим индексы начала и конца совпадения (match), ищем по хранилищу селекторов, вставляем информацию, запускаем поиск далее по строке (а не с её начала) и т.д.

#### Формирование итоговой лог-строки

Разная информация вставляется в лог-строку (не в неизменяемый string, а в StringBuilder) последовательным проходом по outputPattern от начала до конца.

Для String-store-ов: паттерн просматривается посимвольно, учитываются не только селекторы, но и символы помимо них, формируется итоговая строка со вставленной вместо селекторов информация и окружающими селекторы символами.

Для DB-store-ов: важны лишь селекторы, а не символы вокруг них, поскольку формируется не строка, а объект типа, который будет храниться в БД.

### Пакетная обработка

* Позволяет сохранять логи в памяти компьютера при проблемах с хранилищем логов, не тормозя основной поток программы, при нормализации ситуации сохранённые логи поступают на хранение.
* Классы-потомки StoreManager-а с пакетной обработкой имплементируют IBatchProcessingCompatible.
* StoreManager-класс с пакетной обработкой имеет BlockingCollection для хранения очереди задач.
* экземпляр Logger-класса пытается добавить логи на обработку (TryLog()) – по факту добавляет в очередь.
* очередь обрабатывается отдельным фоновым потоком, который запускается в storeManager-потомке.
* каждый поставщик использует очередь => сколько поставщиков с пакетной обработкой, столько и доп. потоков.
* Обоснованно используется именно пул потоков через Task.Factory.StartNew, а не класс Thread – Task-обёртка работает быстрее.

### Фиксация заполненности очереди

* Этот функционал определён классом ProcessingQueueChecker.
* Может быть ситуация, когда потребитель (store) обрабатывает записи на логирование, но очередь уже забита на 100% - скорее всего множество записей просто не добавляется.
* Фиксируется время начала 100% заполненности и при её продолжении идёт счёт времени, это время 100% заполненности пишется в лог, чтобы пользователь мог оценить как долго в потребитель (store) не пишутся логи.
* Крайне маловероятна такая ситуация: очередь заполнена на 100%, но потребитель и производитель идут нога в ногу, записи не теряются. Реализованная логика такое не обрабатывает и будет сообщать о потерях записей, но это не существенно.

### Самодиагностика

* Для логгирования самого себя запрашивается логгер классическим образом, однако в GetLogger необходимо передать true(IsInsideLogger = true, по дефолту оно false). Можно использовать обычные методы взаимодействия как с любым другим логгером. Параметр изменит принцип работы метода Log() в классе Logger и выведет инфу в заданный конфигом storemanager для самологирования логгера (по дефолту это console – она самая устойчивая, заметная и не зависит от доступного пространства на жёстком диске, а коль закончится ОЗУ, так вся программа итак рухнет вместе с логгером).
* Фактически теперь потоки store-ов и main-поток пишут в Log одновременно, а Log синхронный. Однако store-потоки не делают этого часто, поэтому не ожидаются проблемы с производительностью.

### Уникальные Id-шники для БД

* Каждая лог-запись наделяется id-шником для записи в БД.
* При крайне высокой нагруженности таблицы БД для логгера (>1000 записей в секунду) может быть недостаточно использовать ГСЧ для формирования id-шника при одинаковом timestamp-е записей.
* Риски:
  + запуск экземпляров одного приложения (один и тот же конфиг, где указана одна и та же таблица в БД) в разных процессах/доменах, пишущих в одну таблицу БД,
  + слишком интенсивное логирование в рамках одной программы, работающей в единственном экземпляре.
* Формат id-шника: time logLevel <{PID\_HEX}{AppDomainId\_HEX}{SessionIterator\_HEX}>

# Эксплуатация

## Подключение

1. Подключить либу, пробросить зависимость.
2. StruLog.StruLogProvider.Init(configPath); в Program.cs проекта, который использует логгер. Можно вызвать Init(configName, inProjectDir:true), тогда конфиг будет искаться в директории проекта.
3. Создаём static readonly логгеры для каждого класса с указанием логируемого типа:

#logger = LoggersFactory.GetLogger<MainClass>()– каждый логгер привязан к логируемому типу.

#logger = LoggersFactory.GetLogger(typeof(MainClass)) – если MainClass статический.

## Создать кастомный loglevel

Добавить значение loglevel-перечисления (enum), вызвать loggerObj.Log(…) со своим кастомным левелом. + сделать свои методы расширения под него, где уже будут подставлены нужные аргументы, вкл новый loglevel.

Необходимо учесть заданные весовые коэффициенты для уровней:

-trace 100

-debug 200

-info 300

-important 400

-warning 500

-error 600

-fatal 700

Веса используются для ранжира левелов и учета параметра minLogLevel, который указывает нижний порог серьёзности.

## Изменить цвета logLevel-ов

Отредактировать ConsoleSM. SwitchConsoleColorByLogLevel()

## Конфиг-файл

* Загружается и применяется только на старте программы, для вступления в силу изменения конфига необходимо перезапустить программу – это правильно, поскольку ряд параметров внезапно изменившись создадут проблемы.
* Содержание частично регистронезависимое, но лучше не экспериментировать.
* Цвета настраиваются в коде (ибо вряд ли они будут меняться).
* **Конфиг находится на стороне использующего логгер проекта и для каждого проекта-потребителя конфиг независимый.**

### path

Где хранить логи.

Селекторы:

* y – год,
* m – месяц,
* d – день,
* project – текущая runtime-директория проекта.

{project}/Logs/{y}/{m}/NonameProject\_{d}.log

Рекомендуется всегда использовать селектор d, иначе лог будет дозаписываться в уже имеющийся файл логов, т.к. будут генерироваться одинаковые имена файлов логов.

### outputPattern

* для хранилищ File и Console – учитывается каждый знак паттерна.
* для NoSQL хранилищ – учитывается только наличие селекторов, поскольку хранится лог в JSON-виде: отсутствие селектора приведёт к сохранению default-значения типа для соотв. информации.
* {selector}
* поддерживает escape-последовательности

Селекторы:

* msg
* excMsg – сообщение исключения и его тип
* excClassLine – указывается класс и строка (в 2-х верхних фреймах стека исключения: 0 и 1), где произошло исключение (работает только при ловле в catch, при лишь throw-ании экземпляра исключения информация не будет выведена, ибо технически недоступна)
* excStackTrace (делает красную строку, пишет «STACKTRACE»: и выводит его)
* time
* logLevel
* obj
* loggerName – полное название логгера
  + [поддерживается только String-хранилищами, #FileSM, ConsoleSM]loggerName-i – сокращённое название, оно полезно, когда имя логгера – namespace-путь к классу и он слишком длинный, #AccountsDrivingServer.Services.TelegramIncomingReqHandler), i – количество сегментов namespace-а, указанных справа-налево, i=1,..9. Использование этого селектора немного снижает производительность логирования, поскольку сначала идёт отсчёт сегментов имени, а потом подстрока формируется из loggerName-а.

### time

* “UTC”
* “LOCAL” – текущее время системы (как правило зависящее от местного пояса).

### usingStores

Данные названия синхронизированы с константой NAME, которая есть в каждом типе хранилища.

[ "console", "file", "mongoDB" ]

### insideLoggingStore

Здесь необходимо указать имя хранилища логов самодиагностики.

## Добавление store-ов

* Создаётся класс-потомок от StoreManager, DbStoreManager и подобных.
* Необходимо определить в нём константу NAME, по которой будут запрашиваться настройки из конфига.
* Если он реализует пакетную обработку, то он должен имплементировать IBatchProcessingCompatible, что включает в себя:
  + метод запуска пакетной обработки RunBatchProcessing – он должен запускать метод обработки, размещаемый также в этом классе (#Log()),
  + определение параметров очереди и самой очереди,
  + определение массива с задержками между попытками что-то сделать с хранилищем при пакетной обработке.
* Под него нужен отдельный Exception, наследуемый от StruLogException, например StruLogTelegramException.
* В конфиге под новый store также нужен свой объект, а значит нужно также создать класс-наследник от Store.
* Метод Log() или его аналог могут использовать LogsBatchProccessor для пакетной обработки. Как – рассказано в Алгоритмы работы/Вывод лога.

## Изменить параметры пакетной обработки

Для каждого SMа значения параметров независимы.

* AccessAttemptsDelays\_mSeconds (содержится в классе нужного SMа), позволяет определить задержки в миллисекундах между попытками совершить действия с хранилищем (получить доступ),
* ProcessingQueueSize (содержится в классе нужного SMа)

## Изменить название хранилища в конфиге

Константа NAME в типе SM-а. Благодаря этой константе программа связывает параметры в конфиге с типами хранилищ в коде.

## Изменить символы начала-окончания селекторов

SELECTOR\_START\_CHAR и SELECTOR\_END\_CHAR в StoreManager.

## Добавить селекторы и выводимую информацию

Чтением селекторов и формированием лог-файла занимаются следующие типы:

* Logger: Log() формирует объект типа LogData, в который помещает исходные данные для логирования, из них далее будут формироваться все остальные логируемые данные,
* Классы StringStoreManager, DbStoreManager, StoreManager определяют состав селекторов, алгоритм прохода по паттерну и формирование выводимой инфы.
* FileSM.GetInfoByPathSelector() – формирует путь по селекторам.

## Изменить логируемую самодигностическую инфу

* В классах логгера:
  + о заполненности очереди: в классе ProcessingQueueChecker
* В «паттерне выхода» выбранного хранилища самодиагностики.

## Изменить предупредительный порог занятости очереди обработки

После прохождения этого порога начинаются предупреждения о занятости очереди. По дефолту – это 85%.

ProcessingQueueChecker.QUEUE\_WARNING\_OCCUPIED\_CAPACITY\_PERCENT

## Добавить/изменить параметры в конфиге

* ConsoleFileProvider парсит конфиг.
* Тип конфига: класс Config.
* Названия полей типов конфига должны совпадать с аналогами в конфиг-файле (nameof).

Задачи по улучшению

Подкрашивание эл-ов в консоли

* Автоматическое: URL запросов, объекты, исключения…
* Ручное: доступ к настройкам консоли.