Тестовое задание HtmlExtractorTaskApp - консольное приложение для извлечения содержимого статьи страницы по url.

Параметр командной строки - url с адресом статьи. options.xml - файл, с настройками параметров анализатора. Лежит в директории приложения. Пример файла:

```
<?xml version="1.0"?>
<options>
    <option>
           <plainlenghtrate>10</plainlenghtrate>
           <paragraphrate>10</paragraphrate>
           <dotrate>10</dotrate>
           <headrate>20</headrate>
           <blocktyperate>10</blocktyperate>
           <refdensityrate>10</refdensityrate>
           <tagdensityrate>5</tagdensityrate>
           <idclassbadrate>10</idclassbadrate>
           <nsimilaridbadrate>10</nsimilaridbadrate>
           <viscriterio>10</viscriterio>
    </option>
    <option url="russian.rt.com">
           <plainlenghtrate>5</plainlenghtrate>
           <paragraphrate>10</paragraphrate>
           <dotrate>10</dotrate>
           <headrate>20</headrate>
           <blocktyperate>10</blocktyperate>
           <refdensityrate>10</refdensityrate>
           <tagdensityrate>10</tagdensityrate>
           <idclassbadrate>10</idclassbadrate>
           <nsimilaridbadrate>10</nsimilaridbadrate>
           <viscriterio>8</viscriterio>
    </option>
</options>
```

Для отдельного сайта можно задавать отдельные настройки весов для тестов анализатора и значение критерия видимости узла - viscriterio. Узел без атрибута url - настройки по умолчанию для всех сайтов.

Результат сохраняется в текущей папке приложения во вложенных папках, соответствующих url статьи.

Экстрактор содержимого представлен классом WebPageExtractor, использует следующие компоненты:
1) IHtmlSourceProvider - поставщик разметки исходной страницы
2) IDomParser - парсер разметки в DOM
3) IBlockAnalyzer - анализатор статистик узлов DOM дерева
4) ITextFormatter - обработчик финального текста (тут для установления длины строки не более 80 символов)
5) IPlainTextSaver - сохраняет полученный текст на устройство

Описание работы программы:

```
void WebPageExtractor::run(){
    m_pSourceProvider->doRequest(10000);
    if(!m_pSourceProvider->isRequestComplete()){
        qDebug() << "Html reques failed!";
    }
    QString html = m_pSourceProvider->getHtml();
    m_pParser->setHtml(html);
    m_pParser->calcPrimaryStat();
    QVector<PrimaryStatistics> stats;
    m_pParser->getPrimaryStat(stats);
    m_pAnalyzer->analyze(stats);
    QSet<int> visibleBlocks;
    m_pAnalyzer->getVisibleBlocks(visibleBlocks);
    QString plain = m_pParser->getPlainText(visibleBlocks);
    m_pFormatter->format(plain);
    if(!m_pSaver->saveToDevice(plain)){
        qDebug() << "Failed save to file!";
    }
}
```

1) Делаем запрос поставщику исходного текста разметки страницы.
2) Отдаем текст парсеру.
3) Парсер внутри setHtml формирует структуру, описывающую блоки (в основном - div)
4) Считаем парсером первичные статистики блоков
5) Отправляем первичные статистики анализатору
6) Анализатор внутри себя считает дополнительные статистики, проводит тесты блоков и расставляет веса, которые определяют, будет ли отображаться текст этого узла.
7) Идексы видимых блоков отдаются обратно парсеру, который формирует текст извлеченной статьи.
8) Текст проходит финальную обработку форматтером, который разбивает строки, чтобы длина была не больше 80 символов.
9) Полученный текст сохраняется на устройство

Экстрактор зависит от абстракций (интерфейсов), а не от конкретных реализаций компонентов.

Описание компонентов.

1) HtmlSourceProviderFromWeb - реализация IHtmlSourceProvider. Метод doRequest выполняет запрос, ждет его ожидания и сохраняет результат запроса - текст разметки страницы.

2) WebKitHtmlDomParser - реализация DOM парсера с помощью QtWebKit. Решение не оптимальное, но быстро не нашелся хороший html парсер для Qt. QtWebKit - довольно мощный инструмент, решающий широкий круг задач, и в данном случае использовать его как html парсер это как забивать гвоздь с помощью микроскопа. Но этот компонент всегда можно без проблем заменить, когда

найдется подходящий парсер, благо с другими компонентами реализация не связана.

Парсер удаляет узлы, в которых заведомо не может быть решения. Далее для узлов, которые представляют блоки (div, section, article, header, footer) строится дерево с описанием блоков - первичные статистики.

Первичные статистики:

xmlLength - длина XML блока (внешняя)
plainLength - длина отображаемого текста
dotCount - кол-во точек (коррелирует с кол-вом предложений)
paragraphCount - кол-во параграфов
refLength - длина отображаемого текста всех ссылок внутри блока
maxHeadLevel - максимальный уровень тегов <H1>-<H6>, где H1 - 6, H6 - 1
parent - индекс родителя
id - значение атрибута id
classid - id первого класса в атрибуте class
tag - имя тега

Отформатированный текст getPlainText() получается путем установки невидимости для блоков: не прошедших анализ и получения теста от корня.

3) BlockAnalyzer - реализация IBlockAnalyzer, занимается анализом блоков.

Анализ блоков осуществляется на основе материала из статьи на хабре https://habrahabr.ru/company/mailru/blog/200394/. Узлам раздаются веса в зависимости от статистик и в результате в зависимости от веса узла (visibility) он берется видимым или нет.

Для начала рассчитываются дополнительные статистики:

tagDensity - плотность разметки, отношение длины отображаемого текста к длине XML разметки узла. Чем больше длина тегов, тем ниже tagDensity => тем меньше этот узел представляет собой тело статьи
refDensity - плотность ссылок, отношение отображаемого теста ссылок к отображаемому тексту узла. Чем больше ссылок тем выше refDensity => тем меньше этот узел относится к телу статьи.

Далее все узлы проходят тесты:
applyPlainLengthRate() - чем больше отображаемый текст узла, тем больше узел подходит.
applyParagraphRate() - чем больше кол-во параграфов, тем больше узел подходит
applyDotRate() - чем больше кол-во точек (предложений), тем больше подходит узел
applyHeadRate() - чем больше уровень тега <H1>-<H6>, тем больше вероятность, что это заголовок статьи
applyBlockTypeRate() - повышает вес для одних блоков и их детей и понижает для других блоков
applyRefDensityRate() и applyTagDensityRate() - анализирует статистики tagDensity и refDensity, согласно их описанию выше.

applyIdClassBadRate() - понижает вес узлов, если в их атрибутах id, class находит наименования, которые определяют их как скорее всего ненужные
applySimilarIdBadRate() - понижает вес узлов, у которых id начинаются с общей подстроки. пока не протестировано (не реализовано полностью)

Каждый метод теста принимает параметр maxRate - максимальное изменение видимости блока. Это вес теста. Больше вес => больше влияние данного теста.

Данные веса, а также проходной балл visibility задаются через опции

```
void setOptions(IAnalyzerOptions* options);

class IAnalyzerOptions{
public:
    virtual int plainLengthRate()=0;
    virtual int paragraphRate()=0;
    virtual int dotRate()=0;
    virtual int headRate()=0;
    virtual int blockTypeRate()=0;
    virtual int refDensityRate()=0;
    virtual int tagDensityRate()=0;
    virtual int idClassBadRate()=0;
    virtual int similarIdBadRate()=0;
    virtual int visibleCriterio()=0;
};
```

, которые могут браться из файла конфигурации.

4) OutputFormatter - реализация интерфейса ITextFormatter. Разбивает строки, чтобы длина была меньше 80 символов.

5) TextSaverToFile - реализация IPlainTextSaver, сохраняет отформатированный текст статьи в файл, который помещает в созданное им дерево папок, соответствующих URL статьи.

Работа проверялась на следующих статьях:

https://russian.rt.com/article/145367
http://ria.ru/world/20160201/1368123516.html
http://www.vesti.ru/doc.html?id=2714753
http://tass.ru/mezhdunarodnaya-panorama/2628202
http://ren.tv/novosti/2016-02-01/rossiya-vpervye-ispytaet-noveyshie-istrebiteli-su-35s-v-boevyh-usloviyah-v-sirii

На некоторых сайтах требуется подобрать веса методов для тестирования, некоторые сайты не работают из-за кодировки страницы, нужно разбираться. Приведенные выше страницы практически проходят тест на дефолтовых настройках из options.xml и отображают полностью содержимое статьи без лишнего.

Динамические библиотеки в директории с исполняемым фалом относятся к Qt. Так получилось, что при использовании в качестве DOM парсера QtWebKit потянул за собой много зависимых библиотек.