Субъективный объективизм (http://www.scrutator.me/)

(https://twitter.com/ixsci) (http://www.scrutator.me/syndication.axd)

← Добро пожаловать в параллельный мир. Часть 3: Единый и Неделимый (/post/2012/08/28/parallel-world-p3.aspx)

Добро пожаловать в параллельный мир. Часть 1: Мир многопоточный → (/post/2012/04/04/parallel-world-p1.aspx)

Добро пожаловать в параллельный мир. Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

3. июня 2012 ixSci (http://www.scrutator.me/author/Adminaspx) разработка (/category/разработка.aspx) (2) (/post/2012/06/03/parallel-world-p2.aspx#comment)

Продолжая тему начатую в части 1 (http://scrutator.me/post/2012/04/04/parallel-world-p1.aspx), перейдём от простейшего способа использования потоков, к следующему, более продвинутому, способу мульти-поточного программирования, -асинхронному программированию.

> Под простейшим здесь я понимаю не отношение сложность\простота, а тот факт, что потоки, мьютексы и вся та “кухня” из части 1 являются базовыми блоками для всего остального в многопоточном мире.

Под асинхронным программированием я понимаю стиль программирования, который можно охарактеризовать следующим выражением: “дал задачу и забыл”. Т.е. это такой стиль при котором все тяжеловесные задачи исполняются в отличном от вызывающего потоке, и результат выполнения которых может быть получен, вызывающим потоком, когда он того пожелает(при условии, что результат доступен).

К примеру, вам необходимо загрузить большой файл с диска, и отобразить его в интерфейсе пользователя. Вы, конечно, можете сделать это из основного потока, заставив пользователя ждать. Но это не сделает его жизнь счастливее, и довольство вашим софтом постепенно будет падать. Множество софта, которое я встречал, именно так и поступает. Если же идти по пути асинхронного программирования. тогда метод, отвечающий за загрузку файла с диска, не будет блокировать поток выполнения, а будет загружать файл в отдельном потоке, и вернёт результат по первому требованию. Т.е. фактически будет отдана задача: “Загрузи файл”, и больше никаких взаимодействий с этой задачей, до её выполнения, из основного потока, не будет. При этом пользовательский интерфейс будет оставаться отзывчивым всё это время. О том, что есть в C++ для упрощения реализации асинхронного программирования и пойдёт речь в этой статье.

Говорим о будущем

В прошлой статье мы рассмотрели метод построения системы ожидания событий на *std::condtiotion_variable.* Это, без сомнения, очень удобный метод для моделей, в которых есть некий субъект, который **постоянно** ожидает наступления некоторого события. Но существуют и такие ситуации, когда вам необходимо дождаться наступления некоторого события один раз. После чего, оно перестаёт интересовать вас. К примеру, вы стоите в очереди за загранпаспортом и ждёте своей очереди. Благо, в вашем отделении вы получили номерок и теперь ждёте, пока вас вызовут и сообщат, верно ли заполнено ваше заявление. Использование *std::condition_variable* возможно, но было бы избыточным в данной ситуации, т.к. это слишком громоздко заводить *condition_variable + mutex* раде одного уведомления. Необходим некий механизм однократного уведомления. И такой механизм, как вы наверно догадались, существует и именуется он как *future (http://en.wikipedia.org/wiki/Future_(programming)).*

Слова выше, кодом:

```
#include <iostream>
#include <future>

std::future<bool> submitForm(const std::string& form);

int main()
{
    auto check = submitForm("my form");
    if(check.get())
        std::cout << "Wow I've got a passport!\n";
    else
        std::cout << "Dammit, they found a mistake again!\n";
}
```

> Реализация submitForm отсутствует, т.к. на данном этапе мы не обладаем достаточными знаниями для её реализации .

Класс *std::future* представляет собой обертку, над каким-либо значением или объектом(далее значением), вычисление или получение которого происходит отложено (http://ru.wikipedia.org/wiki/Отложенные_вычисления). Точнее, *future* предоставляет **доступ** к некоторому разделяемому **состоянию**, которое состоит из 2-х частей: данные(здесь лежит значение) и флаг готовности. *future* является получателем значения и не может самостоятельно выставлять его; роль *future* пассивна.

> Далее по тексту future может быть использовано вместо понятия *разделяемое состояние,* т.к, по сути, эти понятия можно использовать взаимозаменяемо.

Появление, или же вычисление, значения означает, что разделяемое состояние, содержит требуемое значение и флаг готовности “поднят”. С этого момента значение может быть изъято в любое время без каких-либо блокировок. Объект *future* предоставляет исключительный доступ к значению, когда оно было вычислено. Объект *future* не может быть скопирован, а может быть только перемещен, а это, в свою очередь, дает строгую гарантию программисту, что значение полученное им не может быть испорчено в каком-либо другом месте. В силу своей “отложенной” природы, *future* может быть использовано не только в многопоточном коде, но и в однопоточном его варианте; чем и не преминули воспользоваться разработчики стандарта, но об этом далее.

Для получения значения из *future* предназначен метод *std::future::get*. При этом, поток вызвавший *get* блокируется до вычисления значения. Именно поэтому, мы и говорим об отложенном получении значения, ведь поток, получивший объект *future*, может не сразу блокироваться на нём, для получения значения, а может исполнять всё, что угодно и только придя к точке, когда необходимо получить значения *future,* – получить его. Можно, также, просто подождать появления значения без его непосредственного получения, для этого предназначен метод *std::future::wait.* В примере выше был использован метод *get*, т.к. нас интересует сам ответ(значение), а не просто его появление. Если бы нам нужно было просто получить уведомления, и мы были бы не заинтересованы в ответе мы могли бы использовать метод *wait*. Или же использовать *future<void>* вместо *future<bool>* вкупе с методом *get,* т.к. *future<void>::get* ничего не возвращает.

> Хотя нам никто и не мешает просто игнорировать значение возвращаемое *get,* всё же, это является дурным тоном, игнорировать возвращаемые значения. Не для того они возвращаются.

Метод *wait,* также, может быть полезен в том случае, когда избыточно или некорректно “вечно” ожидать появления значения. *wait* позволяет задать интервал ожидания, таким образом предотвращая “вечное” ожидание. Метод *get,* с другой стороны, не позволяет этого сделать. Что вполне соответствует его семантике: вычислить\получить значение.

Помимо *std::future* существует и расширенный вариант, который представляет совместный доступ к разделяемому состоянию. Эта расширенная версия именуется *std::shared_future. shared_future* позволяет нескольким потокам получать уведомления из одного источника.

В качестве примера из жизни, можно взять объявление о приходе поезда. Здесь одно событие ожидается многими пассажирами, поэтому обычный объект *future* не может быть использован в силу своего исключительного права владения. Зато *shared_future* подходит как нельзя лучше. Но это привлекает и дополнительные издержки связанные с разделяемым доступом, а именно: *shared_future* синхронизирует доступ **только** к future, но не к значению, которое хранится в нём. Т.е. вы гарантировано имеете синхронизацию между всеми вызовами *get*, но не имеете оной при дальнейшем изменении объекта, который был им возвращен. А ведь может быть возвращена и не константная ссылка на объект. Таким образом, вся синхронизация, относящаяся к значению, полученному посредством *get* из *future,* ложится на плечи программиста, что, в целом, логично.

Объект *shared_future* может быть получен из объекта *future*, с помощью метода *std::future::share*, после чего объект *future* становится “пустым”, т.е. лишается своего доступа к разделяемому состоянию и не может быть в дальнейшем использован для ожидания или получения значения из него.

Итак, мы рассмотрели различные варианты ожидания на *future* и получения из него значения. Но как же создать *future*, и как вычислить это самое значение, чтобы оно стало доступно во *future*? Для этого существуют специальные сущности, которые являются поставщиками значения для *future*. Мы рассмотрим их все, поочередно, в следующих параграфах.

Пакуем задачи

В арсенале объектов, помогающих в реализации идиомы асинхронного программирования, в C++11, существует вполне логичная для такого рода программирования сущность, - задача. Ведь именно задача является базовым блоком асинхронного программирования, как мы условились во введении к настоящей статье. В C++ роль задачи выполняет объект класса *std::packaged_task.* Его использование идентично использованию *std::function*, с тем лишь отличием, что *std::packaged_task* содержит и является поставщиком значения для *future*. Таким образом, первым методом, рассмотренным нами, получения *future* является *std::packaged_task::get_future*. При выполнении *std::packaged_task* исполняет код функции, который был передан ей при создании и выставляет значение в *future*, которое, в свой черёд, является возвращаемым значением этой функции.

Реализуем submitForm:

```cpp
std::future<bool> submitForm(const std::string& form)
{
    auto handle = [](const std::string& form) -> bool
    {
        std::cout << "Handle the submitted form: " << form << "\n";
        return true;
    };
    std::packaged_task<bool(const std::string&)> task(handle);
    auto future = task.get_future();
    std::thread thread(std::move(task), form);
    thread.detach();
    return std::move(future);
}
```

В коде выше, мы использовали *packaged_task* в качестве аргумента *thread*, для того, чтобы исполнить задачу в отдельном потоке, т.к. её исполнение в том же потоке, что и вызов submitForm уничтожил бы всю её пользу. Хотя ничего не мешает вам использовать её в том же потоке, т.к. *packaged_task* является типичным функтором. Можно мысленно перенести задачу загрузки файла, из начала статьи, в lambda функцию из кода выше и получить, то о чем я говорил в начале – "Дал задачу и забыл". Программирование на уровне задач является весьма удобным механизмом ускорения отзывчивости софта. Кроме того, задачи являются частью повседневной жизни, а следовательно должны быть интуитивно понятны и в программировании.

Ещё одним интересным методом *packaged_task* является *std::packaged_task::make_ready_at_thread_exit*, который принимает аргументы для вызова *packaged_task*. При выполнении данного метода происходит следующее: выполняется код функции, сохраненной в *packaged_task*, результат сохраняется в разделяемом состоянии, НО флаг готовности будет выставлен только после того, как все деструкторы локальных, для данного потока, объектов будут выполнены. Т.е. прямо перед завершением потока. Давайте исправим код *submitForm*, представив deadlock 

```cpp
std::future<bool> submitForm(const std::string& form)
{
    auto handle = [](const std::string& form) -> bool
    {
        std::cout << "Handle the submitted form: " << form << "\n";
        return true;
    };
    std::packaged_task<bool(const std::string&)> task(handle);
    task.make_ready_at_thread_exit(form);
    return task.get_future();
}
```

Т.к. *main* ждёт появления значения в *future*, чтобы продолжить выполнение, а оно появится только по окончании работы потока, - мы имеем deadlock.

Используйте *packaged_task*, тогда когда вам необходимо выполнить некую функцию асинхронно, и получить результат по окончанию её исполнения. Хотя я использовал только один поток с одной задачей никто не мешает сделать по другому, например очередь из задач, которые исполняются в одном потоке. Это позволит нам сэкономить на количестве потоков, если у нас есть непрерывный поток задач. Скорее всего примерно таким(один поток и много задач) и будет ваше применение *packaged_task*, ведь в C++11 есть куда более элегантный способ выполнить одну задачу в отдельном потоке, но об этом речь пойдёт чуть позже.

Даём обещания

Еще одним поставщиком *future* и значений для него является *std::promise*.

В противовес строго пассивного *future*, *promise* является строго активной сущность. Назначение *promise* есть поставка значения для future. *promise* не может считать значение, которые ей же и было выставлено. Эта пара создает целостную сущность, которая является корнем асинхронного программирования в C++.

Для получения *future*, *promise* содержит специальный метод *std::promise::get_future().* А для выставления значения в разделяемом состоянии есть две функции:

- *set_value* – сохраняет значение в разделяемом состоянии и выставляет флаг готовности
- *set_value_at_thread_exit* - сохраняет значение в разделяемом состоянии, а флаг готовности выставляется после отработки деструкторов всех объектов, локальных по отношению к потоку.

Важно помнить, что значение может быть выставлено только один раз. При попытке повторного выставления вы получить исключение *std::future_error*. Поэтому один раз выставленное значение не может быть изменено, если это не ссылка или указатель, конечно.

Как и *future*, *promise* является исключительно перемещаемым типом, но, в отличие от *future* не обладает разделяемой версией. Поэтому для копирования *promise* придется пользоваться сторонними средствами, например *std::shared_ptr.*

Пример работы с *promise*:

```
#include <future>
#include <thread>
#include <limits>

int main()
{
    auto spPromise = std::make_shared<std::promise<void>>();
```

```
    std::future<void> waiter = spPromise->get_future();
    auto call = [spPromise](size_t value)
    {
        size_t i = std::numeric_limits<size_t>::max();
        while(i--)
        {
            if(i == value)
                spPromise->set_value();
        }
    };
    std::thread thread(call, std::numeric_limits<size_t>::max() - 500);
    thread.detach();
    waiter.get();
}
```

В коде выше, создается поток, который отсчитывает числа с конца и уведомляет ожидающую сторону, что число, переданное в качестве аргумента, найдено. Как вы можете заметить, объект *promise* обернут в *shared_ptr*; как я уже говорил ранее. это необходимо для копирования *promise* в новый поток. Хотя можно было её и переместить туда, но, к сожалению *lambda* не содержит синтаксиса для “перемещающего захвата”, а другими методами делать этого не хотелось. Другой особенностью, которую вы можете наблюдать, является *set_value*, без параметров. Это специальная версия *set_value* для *promise* параметризированной типом *void*. Для остальных типов этот метод принимает один аргумент.

Из кода видно, что *promise* предоставляет больше свободы, чем *packaged_task*, которая ограничена выставлением значения только из возвращаемого значения функции, а, следовательно, только по её завершению. *promise*, по факту, является самым гибким поставщиком значения для *future*.

std::promise следует использовать в местах, где необходим полный контроль над местом выставления значения.

Проводим асинхронные вызовы

Последним методом получения *future* является функция *std::async*. *async* принимает в качестве аргументов функцию, аргументы функции и, опционально, флаг, который влияет на политику вызова *async*. Возвращаемым значением *async* является *future*, значение которого будет выставлено по возвращении функции, и будет иметь значение ей возвращённое. Поведение *async*, зависит от переданных флагов следующим образом:

- *launch::async* – если передан этот флаг, то поведение *async* будет следующим: будет создан объект класса *thread*, с функцией и её аргументами в качестве аргументов нового потока. Т.е. *async* инкапсулирует создание потока, получение *future* и предоставляет однострочную запись для выполнения такого кода(скорее всего реализация будет использовать *packaged_task,*

вместо простой передачи функции в поток, “под капотом”, но я не уверен)

- *launch::deferred* – если передан этот флаг, то имя функции *async* становится несколько не логичным. Т.к. никакого асинхронного вызова не произойдёт. Вместо исполнения функции в новом потоке, она, вместе с аргументами, будет сохранена в *future*(еще одна особенность *future*), чтобы быть вызванными позже. Это позже наступит тогда, когда кто-либо вызовет метод *get*(или *wait,* но не *wait_for!*) на *future*, которое вернул *async*. При этот вызываемый объект выполнится в потоке, который вызывал *get*! Это поведение есть ни что иное, как отложенный вызов процедуры.
- *launch::async* | *launch::deferred* - в этом случае будет выбрано одно из двух поведений описанных выше. Какое из двух? Неизвестно и зависит от имплементации.

Как указано выше, флаг политики является опциональным. Поведение по умолчанию эквивалентно передаче флагов *launch::async* | *launch::deferred* в *async*.

```
#include <iostream>
#include <string>
#include <future>
#include <thread>
#include <chrono>

int main()
{

    std::cout << "Main thread id=" << std::this_thread::get_id() << "\n";
    auto asyncDefault = std::async([]()
    {
        std::cout << "Async default, Threadid=" <<
            std::this_thread::get_id() << "\n";
    });

    auto asyncDeffered= std::async(std::launch::deferred,
        [](const std::string& str)
    {
        std::cout << "Async deffer, Threadid="
            << std::this_thread::get_id() << "," << str << "\n";
    }, std::string("end string"));

    auto asyncDeffered2  = std::async(std::launch::deferred, []()
    {
        std::cout << "Async deffer2, Threadid="
            << std::this_thread::get_id() << "\n";
    });

    auto trueAsync = std::async(std::launch::async, []()
    {
```

```
32          std::cout << "True async, Threadid="
33              << std::this_thread::get_id() << "\n";
34      });
35      std::this_thread::sleep_for(std::chrono::seconds(5));
36      std::cout << "Sleep ended\n";
37      asyncDefault.get();
38      asyncDeffered.get();
39      trueAsync.get();
40  }
```

```
1  Main thread id=7772
2  Async default, Threadid=8744
3  True async, Threadid=9332
4  Sleep ended
5  Async deffer, Threadid=7772,end string
```

Пример был собран и запущен в MSVC2012 RC и как Вы можете видеть из вывода выше:

1. Реализация *async* выбрала *launch::async*, как политику по умолчанию
2. Первый(asyncDeffered) *async* с *launch::deffered* был выполнен в том же потоке, что и *main*
3. Первый *async* с *launch::deffered* был выполнен после того, как отработал *sleep*
4. Второй(asyncDeffered2) не был выполнен т.к. не была вызвана соответствующая функция ожидания, а следовательно и вызова не будет.
5. *async* с явно переданным флагом *launch::async* был выполнен в отдельном потоке

std::async с флагом *launch::async* является удобной заменой *std::packaged_task* и прямого использования *std::thread*. Т.е. есть смысл всегда использовать её в тех местах, где нет явной необходимости в использовании *std::thread* и *std::packaged_task*. Например, когда есть необходимость в создании отдельного потока на каждую задачу.

std::async, с флагом *launch::deffered*, удобно использовать в случае необходимого отложенного вычисления не требовательного к ресурсам, т.е. того которое может быть быстро вычислено в месте его получения.

Ловим исключения

Говоря о возвращаемых значениях, получаемых посредством *future*, из других потоков. Мы совершенно не упомянули об исключениях. А ведь они могут быть брошены в любой момент. Что произойдет с исключением, брошенном в другом потоке? Как узнать, было ли брошено исключение? Об этом мы и поговорим в этом параграфе.

В новом стандарте, специально для сохранения исключения был введен новый тип *std::exception_ptr*. На деле *exception_ptr* является *typedef*'ом на некоторый другой, зависимый от реализации тип. Т.к. реализация *exception_ptr* не стандартизирована, значит для нас, пользователей, этот тип является непрозрачным, а значит говорить о деталях его реализации не имеет смысла, да и попросту вредно.

Но если мы не можем использовать его внутренности, чем же он тогда нам интересен? А интересен он операциями, которые можно производить с объектом *exception_ptr*, точнее операцией – *std::rethrow_exception*. *rethtow_exception* принимает в качестве аргумента *exception_ptr* и бросает исключение, которое *exception_ptr* содержит, при этом *exception_ptr* должен быть не нулевым.

Следующие блоки кода эквивалентны:

```
1  //Как-то сохранаяем исключения типа std::bad_alloc
2  std::exception_ptr storedPtr = ...
3  std::rethrow_exception(storedPtr);
```

и

```
1  throw std::bad_alloc();
```

Существуют 2 метода, с помощью которых можно заполнить многоточие выше.

Например это могло бы выглядеть так:

```
1  std::exception_ptr storedPtr;
2  try
3  {
4      throw std::bad_alloc();
5  }
6  catch (std::exception&)
7  {
8      storedPtr = std::current_exception();
9  }
```

Здесь используется функция *std::current_exception*, которая возвращает *exception_ptr* указывающий на текущее исключение(т.е. исключение, которое в данный момент обрабатывается) или его копию. Если в данный момент нет обрабатываемого исключения, тогда будет возвращен нулевой *exception_ptr*.

Но если нам нужно просто сохранить исключение, а не перехватывать какое-то существующее, то “городить огороды” из *try\catch* вовсе не обязательно, код описанный выше можно записать следующим образом:

```
std::exception_ptr storedPtr = std::make_exception_ptr(std::bad_alloc());
```

Здесь использован второй метод получения *exception_ptr*, этот метод является простой надстройкой над *current_exception* и представлен для удобства пользователей. Так, хорошо, мы теперь можем сохранять исключения и бросать их позже и в другом месте. Но зачем всё это нужно?

А нужно это для проброса исключений между потоками. И происходит это различными способами, в зависимости от поставщика *future.* Но со стороны *future* всё выглядит одинаково: при ожидании на *future* посредством метода *get*, если в *future* был помещен *exception_ptr*, тогда происходит выброс исключения в потоке, в котором происходит ожидание. Если происходит ожидание посредством других методов, тогда флаг готовности в разделяемом состоянии выставляется, НО исключение не бросается:

```
#include <iostream>
#include <future>
#include <exception>

int main()
{
    auto first = std::async([]()
    {
        throw std::bad_alloc();
    });

    auto second = std::async([]()
    {
        throw std::bad_alloc();
    });

    try
    {
        first.get();
    }
    catch(std::exception&)
    {
        std::cout << "catch exception from the first\n";
    }
    second.wait();
    std::cout << "second has been ended\n";
}
```

Кстати, из кода выше помимо вышеописанного поведения можно заметить и то, как прозрачно происходит передача исключения из одного потока в другой. Вы просто бросаете исключение, и поток владеющий *future* получит его незамедлительно в виде исключения брошенного в нем самом! Это

поведение характерно когда используются *std::thread*, *std::packaged_task* и *std::async*.

Весь этот механизм работает следующим образом: помимо значений и отложенных процедур, *future* может хранить еще и *exception_ptr.* Соответственно, когда в функции выбрасывается исключение оно отлавливается одним из вышеназванных примитивов и сохраняется в *future*. Флаг готовности, также, выставляется. *future* дожидается появления флага готовности и проверяет, не сохранен ли *exception_ptr* в разделяемом состоянии, и если это так бросает исключение посредством *std::rethrow_exception*. Вот так вот всё просто и незамысловато. Зато как элегантно!

Особняком, среди поставщиков *future*, снова стоит *std::promise,* т.к. и в этом случае оно отличается большей гибкостью. *promise* содержит 2 метода для задания *exception_ptr* в *future*: *std::promise::set_exception* и *std::promise::set_exception_at_thread_exit*. Они полностью идентичны методам для задания значения, с той лишь разницей, что выставляют исключение в *future*.

```
#include <iostream>
#include <future>
#include <exception>
#include <thread>
#include <memory>

int main()
{
    auto spPromise = std::make_shared<std::promise<void>>();
    auto waiter = spPromise->get_future();
    auto call = [spPromise]()
    {
        spPromise->set_exception(std::make_exception_ptr(std::bad_alloc()));
    };
    std::thread thread(call);
    try
    {
        waiter.get();
    }
    catch(std::exception&)
    {
        std::cout << "catch exception\n";
    }
    thread.join();
}
```

Как вы можете убедиться, работа с исключениями в многопоточной среде мало чем отличается от работы в однопоточной. Что не может не радовать нас, программистов; фактически ничего нового учить здесь не придётся.

Остальные статьи цикла:

Часть 1: Мир многопоточный (http://scrutator.me/post/2012/04/04/parallel-world-p1.aspx)

Часть 3: Единый и Неделимый (http://scrutator.me/post/2012/08/28/parallel-world-p3.aspx)

Часть 4: Порядки и отношения (http://scrutator.me/post/2015/08/14/parallel-world-p4.aspx)

Часть 5: Граница на замке (http://scrutator.me/post/2015/10/15/parallel_world_p5.aspx)

Метки : C++ (http://www.scrutator.me/?tag=C%2b%2b), future (http://www.scrutator.me/?tag=future), promise (http://www.scrutator.me/?tag=promise), packed_task (http://www.scrutator.me/?tag=packed_task), async (http://www.scrutator.me/?tag=async), exception_ptr (http://www.scrutator.me/?tag=exception_ptr)

Текущий рейтинг: 5.0 (7 голосов)

Похожие записи

Добро пожаловать в параллельный мир. Часть 2: Мир асинхронный (/post/2012/06/03/parallel-world-p2.aspx)

Продолжая тему начатую в части 1, перейдём от простейшего способа использования потоков, к следующем

Добро пожаловать в параллельный мир. Часть 1: Мир многопоточный (/post/2012/04/04/parallel-world-p1.aspx)

Ну вот мы и дождались, C++, наконец, перестал игнорировать ситуацию за окном и признал, что “исполня

Добро пожаловать в параллельный мир. Часть 3: Единый и Неделимый (/post/2012/08/28/parallel-world-p3.aspx)

Данная статья является третьей в цикле статей(часть 1 и часть 2) о многозадачности в C++11. По сути,

Последние записи

Вся правда об указателях. Часть 3: Завершающая (/post/2016/03/30/pointers_demystified_p3.aspx)
Рейтинг: 3.5 / 4

Обзор книги Dependency Injection in .NET (/post/2016/03/19/review_dep_injection.aspx)
Нет оценок

Вся правда об указателях. Часть 2: Памятная (/post/2015/12/30/pointers_demystified_p2.aspx)
Рейтинг: 5 / 2

Обзор книги Applied Cryptography (/post/2015/12/25/review_applied_cryptography.aspx)
Рейтинг: 5 / 1

Вся правда об указателях. Часть 1: Вводная (/post/2015/11/26/pointers_demystified_p1.aspx)
Рейтинг: 5 / 3

Обзор книги C# 5.0 in a Nutshell: The Definitive Reference (/post/2015/11/24/review_csharp5_in_a_nutshell.aspx)
Нет оценок

Добро пожаловать в параллельный мир. Часть 5: Граница на замке (/post/2015/10/15/parallel_world_p5.aspx)
Рейтинг: 5 / 2

Список категорий

(/category/feed/книги.aspx) книги (15) (/category/книги.aspx)

(/category/feed/разработка.aspx) разработка (43) (/category/разработка.aspx)

Список записей по годам/месяцам

2011

2012

2013

2014

2015

2016

Март (/2016/03/default.aspx) (2)