Пара 2: Задача про систему контроля версий

1. Разбор задачи про Lazy

Начнём с разбора предыдущего задания. В принципе, половина группы сдала задачу почти сразу же и особых проблем с ней не имела, но есть некоторые тонкости, на которые хотелось бы обратить внимание. Во-первых, для многопоточного режима с синхронизацией все совершенно разумно применили паттерн «Double-checked locking», чтобы если значение не надо пересчитывать, не надо было и делать блокировку. Но большинство сделало так:

```
private T value;

T get() {
    if (value == NONE) {
        synchronized (this) {
            if (value == NONE) {
                value = supplier.get();
            }
        }
    }
    return value;
}
```

Дело в том, что компилятор вправе выполнять оптимизации, да и процессор может изменять порядок инструкций, чтобы ускорять вычисления. Поэтому может так случиться, что внутри `supplier.get();` начнёт вызываться конструктор, выделится память под результат, указатель на выделенную память присвоится в `value`, произойдёт переключение потока, второй поток дойдёт до `if (value == NONE) {`, увидит, что в `value` уже какой-то указатель, не `NONE`, вернёт `value`, вызывающий им воспользуется и упадёт, потому что конструктор ещё не отработал и объект `value` ещё не инициализирован. Не уверен, что такое может произойти конкретно в этом случае, но многопоточные синглтоны так точно делать умели, поэтому Double-checked locking считается плохой практикой.

Это очень легко поправить (правда, поправится только начиная с Java 5, в более ранних версиях не поправить никак):

```
private volatile T value;

T get() {
```

```
    if (value == NONE) {
        synchronized (this) {
            if (value == NONE) {
                value = supplier.get();
            }
        }
    }
    return value;
}
```

Теперь `value` помечен как **`volatile`**, что, с одной стороны, просит компилятор и Java-машину не оптимизировать код, связанный с этим полем, с другой стороны, добавляет memory barrier, заставляя процессор синхронизировать чтение и запись в это поле между ядрами. Теперь сюрпризов, связанных с порядком операций, быть не должно, зато теперь любое чтение-запись — это дорогая операция, поскольку требует от процессора всяких сложных действий и убивает преимущество от многоядерности. Поэтому можно поступить вот так:

```
private volatile T value;

T get() {
    T result = value;
    if (result == NONE) {
        synchronized (this) {
            result = value;
            if (result == NONE) {
                result = value = supplier.get();
            }
        }
    }
    return result;
}
```

Так у нас получается всего одно обращение к **`volatile`**-полю, если `value` проинициализировано, вместо двух в примере выше. Выяснить, насколько оно быстрее будет работать (и будет ли быстрее вообще), оставляется читателю как опциональное упражнение на +2 балла к этой задаче.

Следующая тонкость — это сброс `supplier` в **`null`** после того, как мы получили из него значение и он нам больше не нужен. В случае с однопоточным и **`synchronized`**-вариантами это весьма очевидно (поскольку все обращения к `supplier` и в том и в другом случае выполняются только в одном потоке, то просто присваиваем ему **`null`** и всё), а вот в lock-free случае всё может быть очень плохо:

```
T get() {
    if (value == NONE) {
        if (supplier != null) {
            if (updater.compareAndSet(this, NONE, supplier.get())) {
```

```
                supplier = null;
            }
        }
    }
    return value;
}
```

Так будет гонка между `supplier = null;` и `supplier.get()`, причём, поскольку переключение между потоками должно попасть в точности между `if (supplier != null) {` и вызовом `supplier.get()` строчкой ниже, и при этом другой поток должен аккуратно занулить `supplier` в строке `supplier = null;`, то гонка проявляется очень редко (честно говоря, без модификации программы её вообще не удалось воспроизвести). Это общая проблема всех гонок, программа может вести себя как абсолютно правильно работающая 10 лет, а потом внезапно упасть, и это не поймать ни юнит-тестами, ни ручным тестированием. Поэтому с многопоточными программами (особенно lock-free) надо очень осторожно — знать типовые приёмы, гарантирующие отсутствие проблем, избегать известных ошибок и всегда внимательно относиться к тому, что вы пишете. В данном случае для воспроизведения гонки может быть полезен `Thread.sleep(0);` или, что лучше, `Thread.yield();`. Вообще, семантика программы не должна по определению зависеть от работы планировщика, так что вставка `Thread.yield();` куда угодно не должна никак влиять на то, что делает программа. Это можно использовать для воспроизведения сложных багов.

2. Внутреннее устройство Git

Теперь переходим к следующей задаче, она несколько объёмнее, чем предыдущая, и мы будем к ней потом возвращаться и модифицировать написанный для неё код. Задача эта — написать свою локальную систему контроля версий, наподобие Git, но без работы с удалёнными репозиториями (пока что).

Проще всего было бы сказать «сделайте мне как в Git, только лучше, можно начинать решать задачу на паре», но, мне кажется, имеет смысл рассказать, как Git выполняет подобные функции. Так что сейчас, внезапно, ещё один рассказ про Git, на сей раз про его внутреннее устройство. Имеет смысл посмотреть первоисточники: краткий обзор архитектуры Git в «The Architecture of Open Source Applications»[1] и, что полезнее, но длиннее, глава 10 Git Book[2].

Git, как известно, распределённая система контроля версий, поэтому весь репозиторий вынужден хранить локально и, если мы никуда push-ить не собираемся, как раз представляет собой локальную version control system (VCS), которую надо сделать в этой задаче (пользоваться гитом в решении, естественно, можно только по прямому назначению). Когда мы набираем `git init`, создаётся папка `.git`, где лежит вся информация гитового репозитория. Она имеет следующую структуру:

- **HEAD** — ссылка на текущую ветку, которую зачекаутили в рабочей папке;
- **index** — staging area, то место, где формируется информация о текущем коммите;

[1] http://aosabook.org/en/git.html
[2] https://git-scm.com/book

- **config** — конфигурационные опции гита для этого репозитория;
- **description** — «is only used by the GitWeb program, so don't worry about it» (c) Git Book;
- **hooks/** — хук-скрипты (возможность исполнить произвольный код при каком-то действии типа коммита), про которые мы сейчас не будем и в домашке их поддерживать не надо;
- **info/** — тоже локальные настройки репозитория, сюда можно вписать игнорируемые файлы, которые вы не хотите писать в .gitignore, чтобы их не коммитить;
- **objects/** — самое интересное, тут лежит собственно то, что хранится в репозитории;
- **refs/** — тут лежат указатели на объекты из objects (ветки, как мы увидим в дальнейшем);
- **...** — прочие штуки, которые появляются в процессе жизни репозитория и нам пока не интересны.

Гит вообще появился как набор утилит, которые позволяют быстро сделать систему контроля версий, а не как полноценная система контроля версий, так что у гита, помимо общеизвестных команд, есть и команды, позволяющие напрямую работать с репозиторием и делать с ним вручную ужасные вещи. Сам по себе репозиторий в гите — это просто хеш-таблица, которая отображает SHA-1-хеш файла в содержимое файла, ничего более. Можно класть в неё объекты (даже не обязательно файлы), можно получать. Например, вот так:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack

$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4

$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Создали пустой репозиторий, гит нам создал структуру папок `.git/objects`, пока пустую. Командой `git hash-object` мы положили в репозиторий новый объект — строчку `'test content'`. Ключ `-w` означает, что надо не просто посчитать хеш объекта, но и реально записать его на диск, ключ `--stdin` означает, что содержимое объекта надо получить из входного потока, а не из файла. Вызов этой команды вернул нам SHA-1-хеш того, что получилось, и заодно создал файл на диске с содержимым, положив его в `.git/objects`, в подпапку, называющуюся как первые два символа хеша, и в файл, называющийся как остальные 38 символов хеша.

Как достать то, что мы сохранили, обратно:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Команда `git cat-file` показывает содержимое файла, ключ `-p` говорит определить тип объекта и красиво показать его содержимое.

Уже можно сделать версионный контроль вручную с использованием рассмотренных команд (правда, для этого нам потребуется настоящий файл, версионировать строку, как в предыдущем примере, не интересно):

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30

$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1

$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Каждая новая версия в данном случае хранится как отдельный объект, но всему своё время.