应用篇

效率

1. 使用多线程充分利用 CPU

1) 环境搭建

- 基准测试工具选择，使用了比较靠谱的 JMH，它会执行程序预热，执行多次测试并平均
- cpu 核数限制，有两种思路
  1. 使用虚拟机，分配合适的核
  2. 使用 msconfig，分配合适的核，需要重启比较麻烦
- 并行计算方式的选择
  1. 最初想直接使用 parallel stream，后来发现它有自己的问题
  2. 改为了自己手动控制 thread，实现简单的并行计算
- 测试代码如下

```
mvn archetype:generate -DinteractiveMode=false -DarchetypeGroupId=org.openjdk.jmh -DarchetypeArtifactId=jmh-java-benchmark-archetype -DgroupId=org.sample -DartifactId=test -Dversion=1.0
```

```java
package org.sample;

import java.util.Arrays;
import java.util.concurrent.FutureTask;

import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Fork;
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.Warmup;

@Fork(1)
@BenchmarkMode(Mode.AverageTime)
@Warmup(iterations=3)
@Measurement(iterations=5)
public class MyBenchmark {
    static int[] ARRAY = new int[1000_000_00];
    static {
        Arrays.fill(ARRAY, 1);
    }
    @Benchmark
    public int c() throws Exception {

        int[] array = ARRAY;
```

```
        FutureTask<Integer> t1 = new FutureTask<>(()->{
            int sum = 0;
            for(int i = 0; i < 250_000_00;i++) {
                sum += array[0+i];
            }
            return sum;
        });
        FutureTask<Integer> t2 = new FutureTask<>(()->{
            int sum = 0;
            for(int i = 0; i < 250_000_00;i++) {
                sum += array[250_000_00+i];
            }
            return sum;
        });
        FutureTask<Integer> t3 = new FutureTask<>(()->{
            int sum = 0;
            for(int i = 0; i < 250_000_00;i++) {
                sum += array[500_000_00+i];
            }
            return sum;
        });
        FutureTask<Integer> t4 = new FutureTask<>(()->{
            int sum = 0;
            for(int i = 0; i < 250_000_00;i++) {
                sum += array[750_000_00+i];
            }
            return sum;
        });
        new Thread(t1).start();
        new Thread(t2).start();
        new Thread(t3).start();
        new Thread(t4).start();
        return t1.get() + t2.get() + t3.get()+ t4.get();
    }
    @Benchmark
    public int d() throws Exception {
        int[] array = ARRAY;
        FutureTask<Integer> t1 = new FutureTask<>(()->{
            int sum = 0;
            for(int i = 0; i < 1000_000_00;i++) {
                sum += array[0+i];
            }
            return sum;
        });
        new Thread(t1).start();
        return t1.get();
    }
}
```

2) 双核 CPU（4个逻辑CPU）

```
C:\Users\lenovo\eclipse-workspace\test>java -jar target/benchmarks.jar
# VM invoker: C:\Program Files\Java\jdk-11\bin\java.exe
# VM options: <none>
# Warmup: 3 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: org.sample.MyBenchmark.c

# Run progress: 0.00% complete, ETA 00:00:16
# Fork: 1 of 1
# Warmup Iteration   1: 0.022 s/op
# Warmup Iteration   2: 0.019 s/op
# Warmup Iteration   3: 0.020 s/op
Iteration   1: 0.020 s/op
Iteration   2: 0.020 s/op
Iteration   3: 0.020 s/op
Iteration   4: 0.020 s/op
Iteration   5: 0.020 s/op


Result: 0.020 ±(99.9%) 0.001 s/op [Average]
  Statistics: (min, avg, max) = (0.020, 0.020, 0.020), stdev = 0.000
  Confidence interval (99.9%): [0.019, 0.021]


# VM invoker: C:\Program Files\Java\jdk-11\bin\java.exe
# VM options: <none>
# Warmup: 3 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: org.sample.MyBenchmark.d

# Run progress: 50.00% complete, ETA 00:00:10
# Fork: 1 of 1
# Warmup Iteration   1: 0.042 s/op
# Warmup Iteration   2: 0.042 s/op
# Warmup Iteration   3: 0.041 s/op
Iteration   1: 0.043 s/op
Iteration   2: 0.042 s/op
Iteration   3: 0.042 s/op
Iteration   4: 0.044 s/op
Iteration   5: 0.042 s/op


Result: 0.043 ±(99.9%) 0.003 s/op [Average]
  Statistics: (min, avg, max) = (0.042, 0.043, 0.044), stdev = 0.001
  Confidence interval (99.9%): [0.040, 0.045]


# Run complete. Total time: 00:00:20
```

```
Benchmark           Mode  Samples  Score  Score error  Units
o.s.MyBenchmark.c   avgt        5  0.020        0.001   s/op
o.s.MyBenchmark.d   avgt        5  0.043        0.003   s/op
```

可以看到多核下，效率提升还是很明显的，快了一倍左右

3) 单核 CPU

```
C:\Users\lenovo\eclipse-workspace\test>java -jar target/benchmarks.jar
# VM invoker: C:\Program Files\Java\jdk-11\bin\java.exe
# VM options: <none>
# Warmup: 3 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: org.sample.MyBenchmark.c

# Run progress: 0.00% complete, ETA 00:00:16
# Fork: 1 of 1
# Warmup Iteration   1: 0.064 s/op
# Warmup Iteration   2: 0.052 s/op
# Warmup Iteration   3: 1.127 s/op
Iteration   1: 0.053 s/op
Iteration   2: 0.052 s/op
Iteration   3: 0.053 s/op
Iteration   4: 0.057 s/op
Iteration   5: 0.088 s/op


Result: 0.061 ±(99.9%) 0.060 s/op [Average]
  Statistics: (min, avg, max) = (0.052, 0.061, 0.088), stdev = 0.016
  Confidence interval (99.9%): [0.001, 0.121]


# VM invoker: C:\Program Files\Java\jdk-11\bin\java.exe
# VM options: <none>
# Warmup: 3 iterations, 1 s each
# Measurement: 5 iterations, 1 s each
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: org.sample.MyBenchmark.d

# Run progress: 50.00% complete, ETA 00:00:11
# Fork: 1 of 1
# Warmup Iteration   1: 0.054 s/op
# Warmup Iteration   2: 0.053 s/op
# Warmup Iteration   3: 0.051 s/op
Iteration   1: 0.096 s/op
Iteration   2: 0.054 s/op
Iteration   3: 0.065 s/op
Iteration   4: 0.050 s/op
```

```
Iteration   5: 0.055 s/op


Result: 0.064 ±(99.9%) 0.071 s/op [Average]
  Statistics: (min, avg, max) = (0.050, 0.064, 0.096), stdev = 0.018
  Confidence interval (99.9%): [-0.007, 0.135]


# Run complete. Total time: 00:00:22

Benchmark            Mode  Samples  Score  Score error  Units
o.s.MyBenchmark.c    avgt        5  0.061        0.060   s/op
o.s.MyBenchmark.d    avgt        5  0.064        0.071   s/op
```

性能几乎是一样的

限制

1. 限制对 CPU 的使用

sleep 实现

在没有利用 cpu 来计算时，不要让 while(true) 空转浪费 cpu，这时可以使用 yield 或 sleep 来让出 cpu 的使用权给其他程序

```
while(true) {
    try {
        Thread.sleep(50);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- 可以用 wait 或 条件变量达到类似的效果
- 不同的是，后两种都需要加锁，并且需要相应的唤醒操作，一般适用于要进行同步的场景
- sleep 适用于无需锁同步的场景

wait 实现

```java
synchronized(锁对象) {
    while(条件不满足) {
        try {
            锁对象.wait();
        } catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
    // do sth...
}
```

条件变量实现

```java
lock.lock();
try {
    while(条件不满足) {
        try {
            条件变量.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // do sth...
} finally {
    lock.unlock();
}
```

2. 限制对共享资源的使用

semaphore 实现

- 使用 Semaphore 限流，在访问高峰期时，让请求线程阻塞，高峰期过去再释放许可，当然它只适合限制单机线程数量，并且仅是限制线程数，而不是限制资源数（例如连接数，请对比 Tomcat LimitLatch 的实现）
- 用 Semaphore 实现简单连接池，对比『享元模式』下的实现（用wait notify），性能和可读性显然更好，注意下面的实现中线程数和数据库连接数是相等的

```java
@Slf4j(topic = "c.Pool")
class Pool {
    // 1. 连接池大小
    private final int poolSize;

    // 2. 连接对象数组
    private Connection[] connections;

    // 3. 连接状态数组 0 表示空闲， 1 表示繁忙
    private AtomicIntegerArray states;


    private Semaphore semaphore;
```

```java
    // 4. 构造方法初始化
    public Pool(int poolSize) {
        this.poolSize = poolSize;
        // 让许可数与资源数一致
        this.semaphore = new Semaphore(poolSize);
        this.connections = new Connection[poolSize];
        this.states = new AtomicIntegerArray(new int[poolSize]);
        for (int i = 0; i < poolSize; i++) {
            connections[i] = new MockConnection("连接" + (i+1));
        }
    }

    // 5. 借连接
    public Connection borrow() {// t1, t2, t3
        // 获取许可
        try {
            semaphore.acquire(); // 没有许可的线程，在此等待
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < poolSize; i++) {
            // 获取空闲连接
            if(states.get(i) == 0) {
                if (states.compareAndSet(i, 0, 1)) {
                    log.debug("borrow {}", connections[i]);
                    return connections[i];
                }
            }
        }
        // 不会执行到这里
        return null;
    }
    // 6. 归还连接
    public void free(Connection conn) {
        for (int i = 0; i < poolSize; i++) {
            if (connections[i] == conn) {
                states.set(i, 0);
                log.debug("free {}", conn);
                semaphore.release();
                break;
            }
        }
    }
}
```

3. 单位时间内限流

guava 实现

```
@RestController
public class TestController {

    private RateLimiter limiter = RateLimiter.create(50);

    @GetMapping("/test")
    public String test() {
//        limiter.acquire();
        return "ok";
    }
}
```

没有限流之前

```
ab -c 10 -t 10 http://localhost:8080/test
```

结果

```
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 5000 requests
Completed 10000 requests
Completed 15000 requests
Completed 20000 requests
Finished 24706 requests


Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /test
Document Length:        2 bytes

Concurrency Level:      10
Time taken for tests:   10.005 seconds
Complete requests:      24706
Failed requests:        0
Total transferred:      3311006 bytes
HTML transferred:       49418 bytes
Requests per second:    2469.42 [#/sec] (mean)
Time per request:       4.050 [ms] (mean)
Time per request:       0.405 [ms] (mean, across all concurrent requests)
Transfer rate:          323.19 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.4      0      16
```

```
Processing:     0    4   7.6      0     323
Waiting:        0    3   6.9      0     323
Total:          0    4   7.6      0     323

Percentage of the requests served within a certain time (ms)
  50%      0
  66%      2
  75%      8
  80%      8
  90%     10
  95%     16
  98%     16
  99%     16
 100%    323 (longest request)
```

限流之后

```
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Finished 545 requests


Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /test
Document Length:        2 bytes

Concurrency Level:      10
Time taken for tests:   10.007 seconds
Complete requests:      545
Failed requests:        0
Total transferred:      73030 bytes
HTML transferred:       1090 bytes
Requests per second:    54.46 [#/sec] (mean)
Time per request:       183.621 [ms] (mean)
Time per request:       18.362 [ms] (mean, across all concurrent requests)
Transfer rate:          7.13 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   1.1      0      16
Processing:     0  179  57.0    199     211
Waiting:        0  178  57.6    198     211
Total:          0  179  56.9    199     211

Percentage of the requests served within a certain time (ms)

  50%    199
```

```
  66%    200
  75%    200
  80%    200
  90%    201
  95%    201
  98%    202
  99%    203
 100%    211 (longest request)
```

互斥

1. 悲观互斥

互斥实际是悲观锁的思想

例如，有下面取款的需求

```java
interface Account {
    // 获取余额
    Integer getBalance();

    // 取款
    void withdraw(Integer amount);

    /**
     * 方法内会启动 1000 个线程，每个线程做 -10 元 的操作
     * 如果初始余额为 10000 那么正确的结果应当是 0
     */
    static void demo(Account account) {
        List<Thread> ts = new ArrayList<>();
        for (int i = 0; i < 1000; i++) {
            ts.add(new Thread(() -> {
                account.withdraw(10);
            }));
        }
        long start = System.nanoTime();
        ts.forEach(Thread::start);
        ts.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        long end = System.nanoTime();
        System.out.println(account.getBalance()
                + " cost: " + (end-start)/1000_000 + " ms");
    }
}
```

用互斥来保护

```
class AccountSync implements Account {

    private Integer balance;

    public AccountUnsafe(Integer balance) {
        this.balance = balance;
    }

    @Override
    public Integer getBalance() {
        synchronized (this) {
            return this.balance;
        }
    }

    @Override
    public void withdraw(Integer amount) {
        synchronized (this) {
            this.balance -= amount;
        }
    }
}
```

2. 乐观重试

另外一种是乐观锁思想，它其实不是互斥

```
class AccountCas implements Account {
    private AtomicInteger balance;

    public AccountCas(int balance) {
        this.balance = new AtomicInteger(balance);
    }

    @Override
    public Integer getBalance() {
        return balance.get();
    }

    @Override
    public void withdraw(Integer amount) {
        while(true) {
            // 获取余额的最新值
            int prev = balance.get();
            // 要修改的余额
            int next = prev - amount;
            // 真正修改
            if(balance.compareAndSet(prev, next)) {

                break;
```

```
            }
        }
    }
}
```

同步和异步

1. 需要等待结果

这时既可以使用同步处理，也可以使用异步来处理

1. join 实现（同步）

```java
static int result = 0;

private static void test1() throws InterruptedException {
    log.debug("开始");
    Thread t1 = new Thread(() -> {
        log.debug("开始");
        sleep(1);
        log.debug("结束");
        result = 10;
    }, "t1");
    t1.start();
    t1.join();
    log.debug("结果为:{}", result);
}
```

输出

```
20:30:40.453 [main] c.TestJoin - 开始
20:30:40.541 [Thread-0] c.TestJoin - 开始
20:30:41.543 [Thread-0] c.TestJoin - 结束
20:30:41.551 [main] c.TestJoin - 结果为:10
```

评价

- 需要外部共享变量，不符合面向对象封装的思想
- 必须等待线程结束，不能配合线程池使用

2. Future 实现（同步）

```java
private static void test2() throws InterruptedException, ExecutionException {
    log.debug("开始");
    FutureTask<Integer> result = new FutureTask<>(() -> {
        log.debug("开始");
        sleep(1);
        log.debug("结束");
        return 10;
    });
    new Thread(result, "t1").start();
    log.debug("结果为:{}", result.get());
}
```

输出

```
10:11:57.880 c.TestSync [main] - 开始
10:11:57.942 c.TestSync [t1] - 开始
10:11:58.943 c.TestSync [t1] - 结束
10:11:58.943 c.TestSync [main] - 结果为:10
```

评价

- 规避了使用 join 之前的缺点
- 可以方便配合线程池使用

```java
private static void test3() throws InterruptedException, ExecutionException {
    ExecutorService service = Executors.newFixedThreadPool(1);
    log.debug("开始");
    Future<Integer> result = service.submit(() -> {
        log.debug("开始");
        sleep(1);
        log.debug("结束");
        return 10;
    });
    log.debug("结果为:{}, result 的类型:{}", result.get(), result.getClass());
    service.shutdown();
}
```

输出

```
10:17:40.090 c.TestSync [main] - 开始
10:17:40.150 c.TestSync [pool-1-thread-1] - 开始
10:17:41.151 c.TestSync [pool-1-thread-1] - 结束
10:17:41.151 c.TestSync [main] - 结果为:10, result 的类型:class java.util.concurrent.FutureTask
```

评价

- 仍然是 main 线程接收结果
- get 方法是让调用线程同步等待

3. 自定义实现（同步）

见模式篇：保护性暂停模式

4. CompletableFuture 实现（异步）

```
private static void test4() {
    // 进行计算的线程池
    ExecutorService computeService = Executors.newFixedThreadPool(1);
    // 接收结果的线程池
    ExecutorService resultService = Executors.newFixedThreadPool(1);
    log.debug("开始");
    CompletableFuture.supplyAsync(() -> {
        log.debug("开始");
        sleep(1);
        log.debug("结束");
        return 10;
    }, computeService).thenAcceptAsync((result) -> {
        log.debug("结果为:{}", result);
    }, resultService);
}
```

输出

```
10:36:28.114 c.TestSync [main] - 开始
10:36:28.164 c.TestSync [pool-1-thread-1] - 开始
10:36:29.165 c.TestSync [pool-1-thread-1] - 结束
10:36:29.165 c.TestSync [pool-2-thread-1] - 结果为:10
```

评价

- 可以让调用线程异步处理结果，实际是其他线程去同步等待
- 可以方便地分离不同职责的线程池
- 以任务为中心，而不是以线程为中心

5. BlockingQueue 实现（异步）

```
private static void test6() {
    ExecutorService consumer = Executors.newFixedThreadPool(1);
    ExecutorService producer = Executors.newFixedThreadPool(1);
    BlockingQueue<Integer> queue = new SynchronousQueue<>();
    log.debug("开始");
    producer.submit(() -> {
        log.debug("开始");
        sleep(1);
        log.debug("结束");
        try {
            queue.put(10);
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    consumer.submit(() -> {
        try {
            Integer result = queue.take();
            log.debug("结果为:{}", result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
}
```

2. 不需等待结果

这时最好是使用异步来处理

1. 普通线程实现

```java
@Slf4j(topic = "c.FileReader")
public class FileReader {
    public static void read(String filename) {
        int idx = filename.lastIndexOf(File.separator);
        String shortName = filename.substring(idx + 1);
        try (FileInputStream in = new FileInputStream(filename)) {
            long start = System.currentTimeMillis();
            log.debug("read [{}] start ...", shortName);
            byte[] buf = new byte[1024];
            int n = -1;
            do {
                n = in.read(buf);
            } while (n != -1);
            long end = System.currentTimeMillis();
            log.debug("read [{}] end ... cost: {} ms", shortName, end - start);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

没有用线程时，方法的调用是同步的：

```
@Slf4j(topic = "c.Sync")
public class Sync {

    public static void main(String[] args) {
        String fullPath = "E:\\1.mp4";
        FileReader.read(fullPath);
        log.debug("do other things ...");
    }

}
```

输出

```
18:39:15 [main] c.FileReader - read [1.mp4] start ...
18:39:19 [main] c.FileReader - read [1.mp4] end ... cost: 4090 ms
18:39:19 [main] c.Sync - do other things ...
```

使用了线程后，方法的调用时异步的：

```
private static void test1() {
    new Thread(() -> FileReader.read(Constants.MP4_FULL_PATH)).start();
    log.debug("do other things ...");
}
```

输出

```
18:41:53 [main] c.Async - do other things ...
18:41:53 [Thread-0] c.FileReader - read [1.mp4] start ...
18:41:57 [Thread-0] c.FileReader - read [1.mp4] end ... cost: 4197 ms
```

2. 线程池实现

```
private static void test2() {
    ExecutorService service = Executors.newFixedThreadPool(1);
    service.execute(() -> FileReader.read(Constants.MP4_FULL_PATH));
    log.debug("do other things ...");
    service.shutdown();
}
```

输出

```
11:03:31.245 c.TestAsyc [main] - do other things ...
11:03:31.245 c.FileReader [pool-1-thread-1] - read [1.mp4] start ...
11:03:33.479 c.FileReader [pool-1-thread-1] - read [1.mp4] end ... cost: 2235 ms
```

3. CompletableFuture 实现

```
private static void test3() throws IOException {
    CompletableFuture.runAsync(() -> FileReader.read(Constants.MP4_FULL_PATH));
    log.debug("do other things ...");
    System.in.read();
}
```

输出

```
11:09:38.145 c.TestAsyc [main] - do other things ...
11:09:38.145 c.FileReader [ForkJoinPool.commonPool-worker-1] - read [1.mp4] start ...
11:09:40.514 c.FileReader [ForkJoinPool.commonPool-worker-1] - read [1.mp4] end ... cost: 2369 ms
```

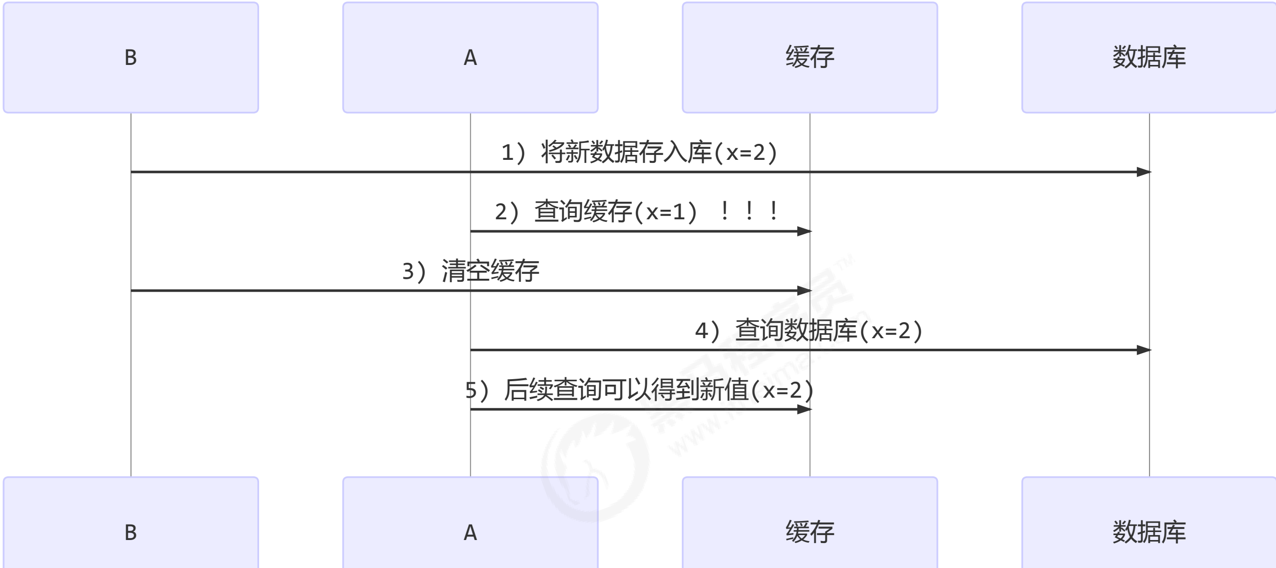
缓存

1. 缓存更新策略

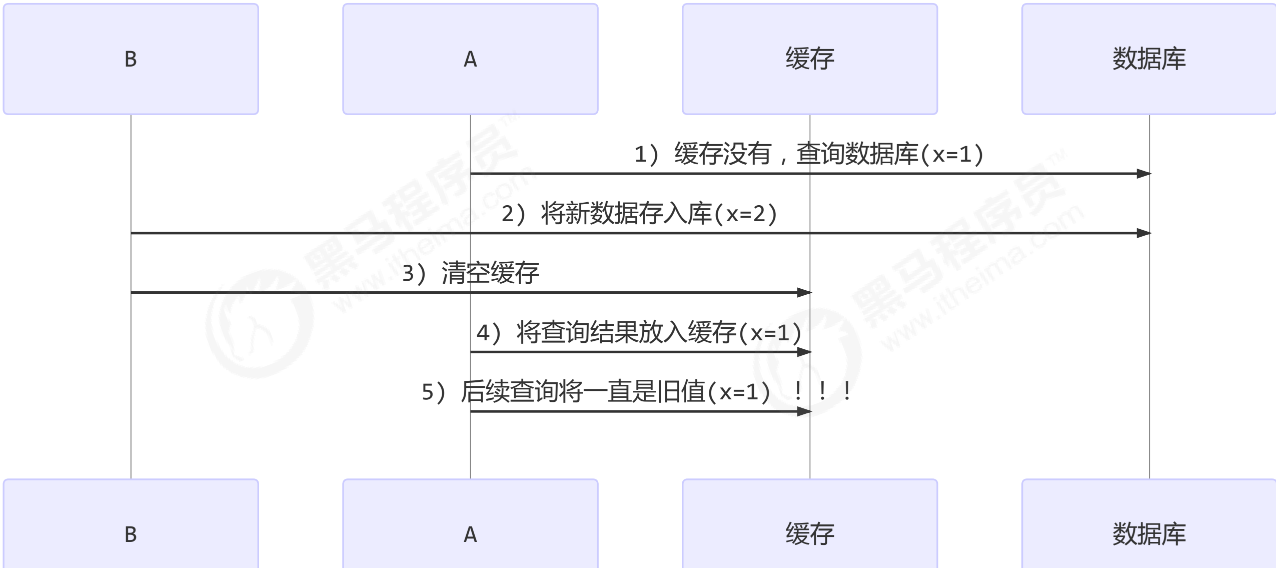
更新时，是先清缓存还是先更新数据库

先清缓存

先更新数据库



补充一种情况，假设查询线程 A 查询数据时恰好缓存数据由于时间到期失效，或是第一次查询



这种情况的出现几率非常小，见 facebook 论文

2. 读写锁实现一致性缓存

使用读写锁实现一个简单的按需加载缓存

```
class GenericCachedDao<T> {
    // HashMap 作为缓存非线程安全, 需要保护
    HashMap<SqlPair, T> map = new HashMap<>();

    ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    GenericDao genericDao = new GenericDao();

    public int update(String sql, Object... params) {
        SqlPair key = new SqlPair(sql, params);

        // 加写锁, 防止其它线程对缓存读取和更改
```

```
        lock.writeLock().lock();
        try {
            int rows = genericDao.update(sql, params);
            map.clear();
            return rows;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public T queryOne(Class<T> beanClass, String sql, Object... params) {
        SqlPair key = new SqlPair(sql, params);
        // 加读锁，防止其它线程对缓存更改
        lock.readLock().lock();
        try {
            T value = map.get(key);
            if (value != null) {
                return value;
            }
        } finally {
            lock.readLock().unlock();
        }

        // 加写锁，防止其它线程对缓存读取和更改
        lock.writeLock().lock();
        try {
            // get 方法上面部分是可能多个线程进来的，可能已经向缓存填充了数据
            // 为防止重复查询数据库，再次验证
            T value = map.get(key);
            if (value == null) {
                // 如果没有，查询数据库
                value = genericDao.queryOne(beanClass, sql, params);
                map.put(key, value);
            }
            return value;
        } finally {
            lock.writeLock().unlock();
        }
    }

    // 作为 key 保证其是不可变的
    class SqlPair {
        private String sql;
        private Object[] params;

        public SqlPair(String sql, Object[] params) {
            this.sql = sql;
            this.params = params;
        }

        @Override
        public boolean equals(Object o) {

            if (this == o) {
```

```java
                return true;
            }
            if (o == null || getClass() != o.getClass()) {
                return false;
            }
            SqlPair sqlPair = (SqlPair) o;
            return sql.equals(sqlPair.sql) &&
                    Arrays.equals(params, sqlPair.params);
        }

        @Override
        public int hashCode() {
            int result = Objects.hash(sql);
            result = 31 * result + Arrays.hashCode(params);
            return result;
        }
    }
}
```

> **注意**
>
> - 以上实现体现的是读写锁的应用，保证缓存和数据库的一致性，但有下面的问题没有考虑
>   - 适合读多写少，如果写操作比较频繁，以上实现性能低
>   - 没有考虑缓存容量
>   - 没有考虑缓存过期
>   - 只适合单机
>   - 并发性还是低，目前只会用一把锁
>   - 更新方法太过简单粗暴，清空了所有 key（考虑按类型分区或重新设计 key）
> - 乐观锁实现：用 CAS 去更新

分治

1. 案例 - 单词计数

```java
private static <V> void demo(Supplier<Map<String, V>> supplier, BiConsumer<Map<String, V>,
List<String>> consumer) {
    Map<String, V> counterMap = supplier.get();
    List<Thread> ts = new ArrayList<>();
    for (int i = 1; i <= 26; i++) {
        int idx = i;
        Thread thread = new Thread(() -> {
            List<String> words = readFromFile(idx);
            consumer.accept(counterMap, words);
        });
        ts.add(thread);
    }

    ts.forEach(t -> t.start());
    ts.forEach(t -> {

        try {
```

```
            t.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });

    System.out.println(counterMap);
}

public static List<String> readFromFile(int i) {
    ArrayList<String> words = new ArrayList<>();
    try (BufferedReader in = new BufferedReader(new InputStreamReader(new FileInputStream("tmp/"
+ i + ".txt")))) {
        while (true) {
            String word = in.readLine();
            if (word == null) {
                break;
            }
            words.add(word);
        }
        return words;
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

解法1：

```
demo(
    () -> new ConcurrentHashMap<String, LongAdder>(),
    (map, words) -> {
        for (String word : words) {
            map.computeIfAbsent(word, (key) -> new LongAdder()).increment();
        }
    }
);
```

解法2：

```
Map<String, Integer> collect = IntStream.range(1, 27).parallel()
    .mapToObj(idx -> readFromFile(idx))
    .flatMap(list -> list.stream())
    .collect(Collectors.groupingBy(Function.identity(), Collectors.summingInt(w -> 1)));
```

2. 案例 - 求和

```
class AddTask3 extends RecursiveTask<Integer> {

    int begin;
    int end;
```

```java
    public AddTask3(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }

    @Override
    public String toString() {
        return "{" + begin + "," + end + '}';
    }

    @Override
    protected Integer compute() {
        // 5, 5
        if (begin == end) {
            log.debug("join() {}", begin);
            return begin;
        }
        // 4, 5
        if (end - begin == 1) {
            log.debug("join() {} + {} = {}", begin, end, end + begin);
            return end + begin;
        }

        // 1 5
        int mid = (end + begin) / 2; // 3

        AddTask3 t1 = new AddTask3(begin, mid); // 1,3
        t1.fork();
        AddTask3 t2 = new AddTask3(mid + 1, end); // 4,5
        t2.fork();
        log.debug("fork() {} + {} = ?", t1, t2);

        int result = t1.join() + t2.join();
        log.debug("join() {} + {} = {}", t1, t2, result);
        return result;
    }
}
```

然后提交给 ForkJoinPool 来执行

```java
public static void main(String[] args) {
    ForkJoinPool pool = new ForkJoinPool(4);
    System.out.println(pool.invoke(new AddTask3(1, 10)));
}
```

结果

```
[ForkJoinPool-1-worker-0] - join() 1 + 2 = 3
[ForkJoinPool-1-worker-3] - join() 4 + 5 = 9
[ForkJoinPool-1-worker-0] - join() 3
[ForkJoinPool-1-worker-1] - fork() {1,3} + {4,5} = ?
[ForkJoinPool-1-worker-2] - fork() {1,2} + {3,3} = ?
[ForkJoinPool-1-worker-2] - join() {1,2} + {3,3} = 6
[ForkJoinPool-1-worker-1] - join() {1,3} + {4,5} = 15
15
```

统筹

案例 - 烧水泡茶

解法1：join

```
Thread t1 = new Thread(() -> {
    log.debug("洗水壶");
    sleep(1);
    log.debug("烧开水");
    sleep(15);
}, "老王");

Thread t2 = new Thread(() -> {
    log.debug("洗茶壶");
    sleep(1);
    log.debug("洗茶杯");
    sleep(2);
    log.debug("拿茶叶");
    sleep(1);
    try {
        t1.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    log.debug("泡茶");
}, "小王");

t1.start();
t2.start();
```

输出

```
19:19:37.547 [小王] c.TestMakeTea - 洗茶壶
19:19:37.547 [老王] c.TestMakeTea - 洗水壶
19:19:38.552 [小王] c.TestMakeTea - 洗茶杯
19:19:38.552 [老王] c.TestMakeTea - 烧开水
19:19:40.553 [小王] c.TestMakeTea - 拿茶叶
19:19:53.553 [小王] c.TestMakeTea - 泡茶
```

解法1 的缺陷：

- 上面模拟的是小王等老王的水烧开了，小王泡茶，如果反过来要实现老王等小王的茶叶拿来了，老王泡茶呢？代码最好能适应两种情况
- 上面的两个线程其实是各执行各的，如果要模拟老王把水壶交给小王泡茶，或模拟小王把茶叶交给老王泡茶呢

解法2：wait/notify

```java
class S2 {
    static String kettle = "冷水";
    static String tea = null;
    static final Object lock = new Object();
    static boolean maked = false;

    public static void makeTea() {
        new Thread(() -> {
            log.debug("洗水壶");
            sleep(1);
            log.debug("烧开水");
            sleep(5);
            synchronized (lock) {
                kettle = "开水";
                lock.notifyAll();
                while (tea == null) {
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                if (!maked) {
                    log.debug("拿({})泡({})", kettle, tea);
                    maked = true;
                }
            }
        }, "老王").start();

        new Thread(() -> {
            log.debug("洗茶壶");
            sleep(1);
            log.debug("洗茶杯");
            sleep(2);
            log.debug("拿茶叶");
            sleep(1);
            synchronized (lock) {
                tea = "花茶";
                lock.notifyAll();
                while (kettle.equals("冷水")) {
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
```

```
                }
            }
            if (!maked) {
                log.debug("拿({})泡({})", kettle, tea);
                maked = true;
            }
        }
    }, "小王").start();
  }
}
```

输出

```
20:04:48.179 c.S2 [小王] - 洗茶壶
20:04:48.179 c.S2 [老王] - 洗水壶
20:04:49.185 c.S2 [老王] - 烧开水
20:04:49.185 c.S2 [小王] - 洗茶杯
20:04:51.185 c.S2 [小王] - 拿茶叶
20:04:54.185 c.S2 [老王] - 拿(开水)泡(花茶)
```

解法2 解决了解法1 的问题，不过老王和小王需要相互等待，不如他们只负责各自的任务，泡茶交给第三人来做

解法3：第三者协调

```
class S3 {
    static String kettle = "冷水";
    static String tea = null;
    static final Object lock = new Object();

    public static void makeTea() {
        new Thread(() -> {
            log.debug("洗水壶");
            sleep(1);
            log.debug("烧开水");
            sleep(5);
            synchronized (lock) {
                kettle = "开水";
                lock.notifyAll();
            }
        }, "老王").start();

        new Thread(() -> {
            log.debug("洗茶壶");
            sleep(1);
            log.debug("洗茶杯");
            sleep(2);
            log.debug("拿茶叶");
            sleep(1);
            synchronized (lock) {
                tea = "花茶";
                lock.notifyAll();
            }
```

```
        }, "小王").start();

        new Thread(() -> {
            synchronized (lock) {
                while (kettle.equals("冷水") || tea == null) {
                    try {
                        lock.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                log.debug("拿({})泡({})", kettle, tea);
            }
        }, "王夫人").start();

    }
}
```

输出

```
20:13:18.202 c.S3 [小王] - 洗茶壶
20:13:18.202 c.S3 [老王] - 洗水壶
20:13:19.206 c.S3 [小王] - 洗茶杯
20:13:19.206 c.S3 [老王] - 烧开水
20:13:21.206 c.S3 [小王] - 拿茶叶
20:13:24.207 c.S3 [王夫人] - 拿(开水)泡(花茶)
```

定时

1. 定期执行

如何让每周四 18:00:00 定时执行任务？

```
// 获得当前时间
LocalDateTime now = LocalDateTime.now();
// 获取本周四 18:00:00.000
LocalDateTime thursday =
now.with(DayOfWeek.THURSDAY).withHour(18).withMinute(0).withSecond(0).withNano(0);
// 如果当前时间已经超过 本周四 18:00:00.000， 那么找下周四 18:00:00.000
if(now.compareTo(thursday) >= 0) {
    thursday = thursday.plusWeeks(1);
}

// 计算时间差，即延时执行时间
long initialDelay = Duration.between(now, thursday).toMillis();
// 计算间隔时间，即 1 周的毫秒值
long oneWeek = 7 * 24 * 3600 * 1000;


ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);
```

```
System.out.println("开始时间：" + new Date());
executor.scheduleAtFixedRate(() -> {
    System.out.println("执行时间：" + new Date());
}, initialDelay, oneWeek, TimeUnit.MILLISECONDS);
```