Jsweet407

April 13, 2024

Contents

1 概述

JSweet407 是一个 Java2JS 的转译器实现，在 Jsweet 的基础上，增加了对 Java 并发特性的支持。JSweet407 主要分为两个部分，转译器扩展与 JS 多线程运行时。其中所支持的特性包括 Thread 类, Runnable 接口, synchronized 关键字等。此外，还支持部分标准类库中的线程方法如 join，sleep 等。基于当前的设计架构，还可以方便地实现 Java 标准库中的各种锁类，我们已在项目中实现了如 ReentrantLock, ReadWriteLock, StampedLock 等类作为示例。本章节将从设计目标、设计思路、示例代码三个部分对 JSweet407 进行阐述。

1.1 设计目标

JSweet407 的设计目标是通过截获跨线程共享变量的读取与写入操作，利用消息传递机制 (postMessage 和 sharedArrayBuffer) 模拟内存共享的读写。JSweet407 不支持 Java 源码的并发模式进行识别、分类，只提供 JS Worker 的并发能力实现方式。另一方面，JSweet407 的多线程运行时系统提供了一系列基础的消息通信、同步与阻塞机制，可以为上层的 synchronized、lock 等提供支持。同时多线程运行时会基于 Java 内存模型对消息的数量进行优化，仅在必要的时候同步数据，以达到最小通信开销。

1.2 主要模块的设计思路

1.2.1 运行时的多线程模型

JSweet407 将所有子线程编译成 JS Worker，每个 Worker 本地都会保留一份所需的共享内存，而主线程会保留所有的共享内存数据作为主内存，并且在 Worker 需要的时候进行数据同步与更新。我们在实现共享内存的更新时，发现有推送和拉取两种方式。其中推送更新是指当某个 worker 产生写操作的时候，会将写的数据推送给另一个 worker；而拉取更新是指当某个 worker 产生读操作的时候，会主动去获取最新的数据。其中推送更新的难点在于 JS runtime 的单线程异步特性，推送更新产生的消息会放到 worker 的事件队列中，这种情况下如果不对程序做全面异步化或者 CPS 变换，将难以实现锁机制以及其他同步机制。因此，JSweet407 的数据同步采用拉取方式，在有必要发生数据同步的阶段，子线程将会进行数据拉取请求，并且阻塞自身直到数据同步完成。

1.2.2 并发类的识别

为了简化工作突出重点,当前 JSweet407 只将直接继承了 Thread 或者直接实现了 Runnable 的类识别为并发类，并不考虑更深层次的继承和实现。

1.2.3 截获跨线程共享变量的读取与写入

JSweet407 在第一趟扫描得到所有并发类之后，会开始分析并发类内部的逃逸变量，并将所有逃逸变量标记为跨线程共享变量，并将相应的类也标记为跨线程类。后续对这些跨线程共享变量编译完成之后，会利用 JS 的 Proxy 特性，在每一个子线程中拦截共享数据的读写，并在需要的时候利用多线程运行时进行数据的更新与拉取。

```java
public void run() {
    while(true){
        synchronized (Desk.lock){
            if(Desk.count==0){
                break;
            }else{
                if(Desk.food_flag==0){
                    try {
                        //桌上无食物，顾客被阻塞
                        Desk.lock.wait();
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }else{
                    System.out.println("客人吃饭");
                    Desk.count--;
                    System.out.println(
                        "还要吃"+Desk.count+"碗");
                    Desk.food_flag=0;
                }
                Desk.lock.notifyAll(); //唤醒厨师
            }
        }
    }
}
```

(a) 示例代码

```java
public run() {
    while((true)) {{
        Comm.sync((Desk.lock_$LI$()));{
            if (Desk.count === 0){
                Comm.unsync((Desk.lock_$LI$()));break;
            } else {
                if (Desk.food_flag === 0){
                    try {
                        Desk.lock_$LI$().wait();
                    } catch(e) {
                        ...
                    }
                } else {
                    console.info("\u5ba2\u4eba\u5403\
                        u996d");
                    Desk.count--;
                    console.info("\u8fd8\u8981\u5403" +
                        Desk.count + "\u7897");
                    Desk.food_flag = 0;
                }
                Desk.lock_$LI$().notifyAll();
            }
        }Comm.unsync((Desk.lock_$LI$()));;
    }};
}
```

(b) 编译结果

1.2.4 消息通信、同步与阻塞机制

JSweet407 利用 postMessage 进行消息的通信，利用 sharedArrayBuffer 和 Atomic 进行线程同步与阻塞行为。当一个 Worker 需要同步数据时，会创建一个相应的 buffer，利用 postMessage 将 buffer 传给数据线程，然后利用 Atomic.wait 阻塞自身直到 buffer 中被写入想要的数据。

JSweet407 对 synchronized 中相同锁的处理依赖于每个锁对象的 key 生成。当多个 worker 中使用了同一把锁并尝试获取时，会向主线程询问这把锁的 key 是否已经上锁，并且阻塞自身直到可以主线程通知上锁。

1.3 示例代码

如Figure 1a所示，这是一段 Java 的多线程带锁的生产者消费者代码。其中 synchronized 用到了外部 Desk 类的静态变量 lock 作为锁对象。而Figure 1b是相应的编译结果 TS 代码。其中 synchronized 被编译成 Comm.sync 方法调用，并且在 synchronized 控制流退出的位置如 break，return 和 continue 等语句处，会调用 Comm.unsync 方法释放相应的锁。sync 和 unsync 都会传入相应的锁对象，JSweet407 的转译器和运行时同时保证了传入的锁对象都具备多线程唯一的 key 值，在锁调度的时候会根据此 key 值进行加锁、解锁以及线程调度等操作。此外，运行时为每个 Object 定义了锁相关的方法如 wait、notify 等方法，

1.4 验收方式

首先要确保命令行环境内有 Python，JDK，TSC 等必要工具与依赖。JSweet407 的所有测试用例都放在项目根目录下的 benchmark 文件夹内。我们提供了一个 bash 脚本用于编译文件查看结果。run.sh 脚本会调用 artifacts 目录下我们已经编译好的 jar 包对 example 文件夹里的文

件进行编译，注意 example 文件夹里一次只能放一个文件否则可能会引起冲突。run.sh 会将编译中间结果放到 tsout 文件目录下，并且将最终编译出来的 js 代码拷贝到 runtime/src/compiled.js 中，使用者可以使用 http 服务器访问 runtime/src/test.html 查看代码运行结果。具体过程可以查看我们制作的验收视频。

2 具体实现

2.1 运行时设计

2.1.1 并发类初始化

并发类 WebWorker 是 Java 中 Thread 类的对应物，同时，也是访问运行时一系列 api 的入口。如果一个类继承了 WebWorker，它就会具备并发类的特性，WebWorker 会将该类中的 source 作为可执行的函数，交给一个初始化好的 Worker 去执行。（注意这里的 Worker 是真正的 Web Worker，而我们的 WebWorker 类只是对 Web Worker 的一个封装。

接下来说明 WebWorker 具体是怎么工作的

```
class WebWorker {
    worker = null;
    workerId = null;
    static workerCounter = 0;
    static nextId() {
        WebWorker.workerCounter += 1;
        return WebWorker.workerCounter;
    }
    init() {
        if (this.worker)
            return;
        // initialize worker
        this.worker = new Worker('./initWorker.js');
        this.worker.onmessage = onmessage;
        // assign the id to every worker
        this.workerId = WebWorker.nextId();
    }
    // 注册完成后，启动worker
    start() {
        this.init();
        this.worker.postMessage({ 'command': 'start', 'source': `(${this.source.
            toString()})()` });
    }
}
```

可以看到，WebWorker 在调用 init 方法后，会自动创建一个 web worker，这个 worker 是一个模版 worker，这意味着所有 WebWorker 都会创建同一个 initWorker，这个 worker 会监听来自主线程的消息。那什么时候开始运行每个线程各自的代码呢？在调用 start 方法后，主线程会将该 worker 需要执行的代码发送给 worker，worker 在收到消息后，就会开始执行。

2.1.2 消息通信

因为每一个 worker 的内存是隔离的，而 worker 之间可能有需要共享的数据，这时就需要引入消息通信的机制。

在介绍我们正在使用的机制之前，先介绍一下我们曾经尝试过的另一种方法。

首先就是使用去中心化的消息传递模型，也就是说，每一个 worker 都可以和另一个 worker 直接通信。每一个 worker 都具有所有共享变量的拷贝，当一个 worker 修改共享变量时，就会通知所有其他持有该共享变量的 worker，以尽可能得保证各个 worker 中的数据的一致性。

这种方法的问题在于，锁的实现会非常麻烦，因为需要在所有相关的 worker 之间达成一致来获取和释放锁，这会涉及大量的消息通信开销。并且因为消息通信具有滞后性，难以保证各个进程之间进行同步。

我们目前采取的方案是：主线程作为消息的中心。也就是说：

1. 所有线程发送消息都经过主线程。
2. 并且，主线程管理所有共享变量，worker 中的本地拷贝主要用于优化消息通信的次数（下文会提到）

消息的格式 消息有非常多的属性，它们通常都有一个 command 属性，command 是让消息的接收者做出行动的指令。比如说下面这条消息：

```
postMessage({ 'command': 'update', key, value \* other properties *\});
```

它是从子线程发送给主线程的数据更新指令，表示该线程已经改变了共享变量的值，并希望其他线程也能看到该更新。除了这个指令，还有很多其他指令，如对共享变量的 query，向主线程通知释放锁等等，后面会介绍。

数据同步的时机 那么在何时同步这些共享变量呢？在这里我们使用了 JavaScript 的**代理对象**。

```
new Proxy(target, {
        get: function (_target, propKey) {
            // 向主线程请求数据
            return _target[propKey];
        },
        set: function (_target, propKey, newValue) {
            // 向主线程发送最新的数据
            return true;
        }
    });
```

每一个共享变量都需要创建一个代理对象，然后对共享变量的操作都需要经过代理对象。这样，当进行对共享变量的 get 操作时，就会转化为对主线程的 query 操作；对共享变量的 set 操作就会向主线程发起 update 操作。

同步消息 在 JavaScript 中，postMessage 是一个**异步方法**，线程发送完消息后就继续执行，而不会等待对方的回复。这在很多情况下是不可忍受的，比如说在子线程对主线程 query 数据时，需要等待主线程回复数据后，拿到这个最新的数据，再继续执行。更多地，在后续的锁的实现，也需要类似的同步机制。

同步的机制的实现如下:

```
static synchronizePostMessage(message) {
    const lock = createLock();
    postMessage({ ...message, lock });
    Atomics.wait(lock, 0, 0);
}
function createLock() {
    const lock = new Int32Array(new SharedArrayBuffer(8));
    return lock;
}
```

这里需要使用到 JavaScript 的 SharedArrayBuffer 和 Atomics 操作，SharedArrayBuffer 是开辟一块共享内存，而 Atomics 可以对共享内存进行原子操作，它们通常组合起来使用。当子线程向主线程发消息时，可以附带一个 SharedArrayBuffer，然后等待这个 SharedArrayBuffer 被修改。

```
function releaseLock(lock) {
    Atomics.store(lock, 0, 1);
    Atomics.notify(lock, 0);
}
```

而在主线程这边，收到 SharedArrayBuffer 后，会在完成操作后对 SharedArrayBuffer 进行写操作，这样子线程就会看到修改，然后停止等待，继续执行。

这就是最基本的同步方法，也是后续很多同步方法的基础。

2.1.3 同步方法

synchroized 在章节示例代码中我们可以看到,如Figure 1a类似的代码会被编译成如Figure 1b形式。Comm 是初始化 worker 时创建的静态类，用于提供 sync，unsync 等方法，调用 Comm 中的方法会向主线程发消息，主线程收到消息后进行相应处理。在实现 synchroized 时，我们定义了以下几个数据结构用以辅助锁的实现, 如下。

```
const blockQueues = new Map();
const lockHolders = new Map();// 锁的持有者
const waitingQueues = new Map();// 锁的等待队列
```

下面是主线程的处理逻辑。

sync(lock) 进程申请锁，如果有其他线程持有锁，将自己加入阻塞队列停止执行，否则线程拿到锁继续执行，代码如Figure 2。

unsync(lock) 进程退出锁，判断锁的重入数是否为 0，若为 0，则为其他阻塞的进程分配锁，代码如Figure 3。

wait() 进程调用 wait 方法进入等待队列，等待不同于阻塞，阻塞为进程未申请到锁，进程被动阻塞，等待为进程等待特定条件发生时主动等待。当进程调用 wait 时会释放自己的锁，以便其他阻塞的线程可以得到调度，代码如Figure 4。

```
if(lockHolders.get(key)){ //判断是否有线程持有该锁
    if(lockHolders.get(key).workerId !== data.workerId)      { //判断持有锁的线程
        是否为自己
        joinBlockQueue(key,data); //加入阻塞队列
    }else{ //实现可重入锁
        lockHolders.get(key).count += 1;
        releaseLock(data.lock); //线程拿到锁, 继续执行
    }
}else{  //当前无线程持有锁
    lockHolders.set(key,data); 设置锁的持有者为本线程
    releaseLock(data.lock);
}
```

Figure 2: sync 逻辑

notify() 进程调用 notify() 方法用来告知其他在相同锁上等待的线程等待条件已满足，并将其中一个从等待队列中移到阻塞队列，等待被调度，代码如Figure 5。

notifyAll() 与 notify() 方法类似，唯一不同点在于将所有该锁的等待队列中所有线程移到阻塞队列。

ReentrantLock 在初始换时会加载 ReentrantLock 类，以便支持使用了 ReentrantLock 锁的代码运行，该类实现了如下方法，当调用对应方法时会向主线程发消息，主线程进行对应操作。

```
class ReentrantLock{
    lock(){...}
    unlock(){...}
    tryLock(){...}
    tryLock(time,TimeUnit){...}
    newCondition(){...}
}
```

ReentrantLock 中的 lock 和 unlock 与 sync 和 unsync 类似, ReentrantLock 本身没有提供 notify() 和 wait() 方法，等待唤醒机制是通过 Condition 类实现的，ReentrantLock 中提供了 newCondition() 方法用来创建一个 Condition。Condition 中提供了 await(), signal(), signalAll() 方法实现等待唤醒。使用 Condition 实现等待唤醒更加灵活，一个锁可以创建多个 Condition，每个 Condition 中的 signal() 和 signalAll() 只会唤醒对应 Condition 的 await()。

主线程处理逻辑如下

await() 为了实现不同条件的等待唤醒，将 synchronized 中的等待队列换成了条件等待队列，其余与 wait() 类似。

signal() 将一个线程从锁的条件等待队列取出，放到阻塞队列, 与 notify() 类似。

signalAll() 将同一锁并且为同一条件的所有线程从等待队列中取出，放到阻塞队列, 与 notifyAll() 类似。

```
if (lockHolders.get(key)) {
    lockHolders.get(key).count -= 1;//持有锁的数量减1
    if (lockHolders.get(key).count === 0) {//判断是否释放锁
        lockHolders.delete(key);
        let dataList = blockQueues.get(key);//取出阻塞队列的第一个元素
        if(!dataList){
            return;
        }
        let first = dataList.shift();
        if(!first){
            return;
        }
        releaseLock(first.lock);
        lockHolders.set(key, first);
    }
}
```

Figure 3: unsync 逻辑

```
function wait(data) {
    let key = data.key;
    data.count = lockHolders.get(key).count; // 记录锁重入数
    joinWaitingQueue(key, data);
    lockHolders.delete(key);//释放锁
    dispatchLock(key); //调度其他进程
}
```

Figure 4: wait 逻辑

tryLock() 进程尝试申请锁，如果当前锁被其他线程持有，返回 false 否则返回 true，代码如Figure 6。

tryLock(time,TimeUnit) time 为时间，TimeUnit 为时间单位，进程在规定时间内尝试申请锁，申请锁失败，加入阻塞队列，超时仍未申请到锁则返回 false，否则返回 true。

其他方法均是在子线程中发送特定消息，然后再主线程中进行处理，而本方法不同，本方法在子线程中需要额外处理，子线程需要将已经超时的进程进行标记，以防再被调度。子线程处理逻辑如Figure 7, 主线程处理逻辑如Figure 8。

ReentrantReadWriteLock 在初始化时会加载 ReentrantReadWriteLock 类，实现了 readLock() 和 writeLock() 方法，返回读锁和写锁

readLock 和 writeLock 类中有如下方法，触发机制与 ReentrantLock 中类似。

```
class writeLock{
    lock(){...}
    unlock(){...}
    tryLock(){...}
```

```
function notify(data) {
    let key = data.key;
    if (waitingQueues.has(key)) {
        let set = waitingQueues.get(key);
        const d = set.shift();
        if (d) {
            joinBlockQueue(key, d);
        }
}
```

Figure 5: notify 逻辑

```
function tryLock(data){
    let key = data.key;
    let type =data.type;
    if(lockHolders.get(key)){ //判断是否有线程占有锁
        if(lockHolders.get(key).workerId !== data.workerId){ //判断占有锁的线程是
            否为本线程
            failReleaseLock(data.lock);//获取锁失败, 返回false
        }else{//本线程已经占有锁, 重入数加一
            lockHolders.get(key).count += 1;
            releaseLock(data.lock);
        }
    }else{//无线程占有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
}
```

Figure 6: tryLock 方法逻辑

```
    tryLock(time,TimeUnit){...}
    newCondition(){...}
}
```

lock() 由于读锁是共享锁，一个进程持有了读锁后其他进程还可以申请到读锁，所以在申请锁时要加额外判断判断逻辑如Figure 9。

unlock() 释放锁时，如果释放的是写锁，并且阻塞队列中第一个请求是读锁请求，则释放所有读锁请求，判断逻辑如Figure 10。

StampedLock 初始化时加载 StampedLock 类，实现了如下几个方法, 触发方式和上述锁类似，先向主线程发消息，具体处理逻辑在主线程。

```
class StampedLock{
    readLock() {...}
    unlockRead(stamp) {...}
```

```
tryLock(time, timeUnit) {
    let waitTime = time * timeUnit;
    const lock = createLock();
    postMessage({ 'command': 'sync', 'key': this.__key, 'lock': lock, "workerId":
        workerId });
    if (Atomics.wait(lock, 0, 0, waitTime) === "timed-out") {
        Atomics.store(lock, 0, 2); //如果超时, 将锁标记为超时
        return false;
    }
    return true;
}
```

Figure 7: 带参数的 tryLock 子线程处理逻辑

```
function tryLock(data){
    let key = data.key;
    let type =data.type;
    if(lockHolders.get(key)){ //判断是否有线程占有锁
        if(lockHolders.get(key).workerId !== data.workerId){ //判断占有锁的线程是
            否为本线程
            joinBlockQueue(key,data);//加入阻塞队列
        }else{//本线程已经占有锁, 重入数加一
            lockHolders.get(key).count += 1;
            releaseLock(data.lock);
        }
    }else{//无线程占有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
}
```

Figure 8: 带参数的 tryLock 主线程处理逻辑

```
    writeLock() {..}
    unlockWrite(stamp) {...}
    tryReadLock(time, timeUnit) {...}
    tryWriteLock(time, timeUnit) {...}
    tryOptimisticRead() {...}
    validate(stamp) {...}
    tryConvertToWriteLock(stamp) {...}
    tryConvertToReadLock(stamp) {...}
}
```

StampedLock 中使用 state 变量标识当前锁的版本号以及锁的信息。

state 是一个整型变量，对应二进制的前 24 位用来表示版本号，后八位的高一位用来表示是否有写锁，低七位用来表示读锁的个数，因此最多 127 个读锁。

```
if(type === 'Write'){ //新申请的锁是写锁
    if(lockHolders.get(key)){ //当前有线程持有锁
        if(lockHolders.get(key).workerId !== data.workerId || lockHolders.get(key
            ).type === 'Read'){ //如当前锁的持有者不是本线程或者，本线程已经持有读
            锁
            joinBlockQueue(key,data); //加入阻塞队列
        }else{
            lockHolders.get(key).count += 1;//重入数加一
            releaseLock(data.lock);
        }
    }else{//无线程持有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
    }
}else{  //新申请的是读锁
    if(lockHolders.get(key)){ //有线程持有锁
        if(lockHolders.get(key).workerId !== data.workerId && lockHolders.get(key
            ).type === 'Write')    {  //如果是其他线程的写锁
            joinBlockQueue(key,data); //加入阻塞队列
        }else{
            lockHolders.get(key).count += 1; //重入数加一
            releaseLock(data.lock);
        }
    }else{ //无线程持有锁
        lockHolders.set(key,data);
        releaseLock(data.lock);
    }
}
```

Figure 9: 读写锁判断逻辑

readLock()　没有写锁且读锁数量小于 127 时申请成功，返回 state 值，否则返回 0，表示申请失败。

writeLock()　没有任何其他锁时申请成功，返回 state 值，否则返回 0，表示申请失败。

unlockRead(stamp)　根据 stamp 值释放读锁，并更新 state 值。

unlockwite(stamp)　根据 stamp 值释放写锁，并更新 state 值。

tryReadLock()　尝试申请读锁，若有其他进程持有写锁则返回 0，否则返回 state。

tryWriteLock()　尝试申请写锁，若有其他进程持有读或者写锁锁则返回 0，否则返回 state。

tryReadLock(time, TimeUnit)　在给定时间内申请读锁，申请成功返回 state，申请失败返回 0。

```
//如果阻塞队列第一个是申请读锁，那么将所有读都释放
if(first.type === 'Read'){
    //从后向前遍历，释放所有读锁
    for(let i = dataList.length - 1; i >= 0; i--){
        if(dataList[i].type === 'Read'){
            releaseLock(dataList[i].lock);
            lockHolders.get(key).count += 1;
            dataList.splice(i,1);
        }
    }
}
```

Figure 10: 读写锁释放逻辑

tryWriteLock(time, TimeUnit) 在给定时间内申请写锁，申请成功返回 state，申请失败返回 0。

tryOptimisticRead() 尝试乐观读，返回一个 state 值。

validate(stamp) 判断当前版本是否改变（是否有线程申请过写锁），若版本改变，返回 0，否则返回 1。

tryConvertToWriteLock(stamp) 读锁尝试转换为写锁，失败返回 0，成功返回 state。

tryConvertToReadLock(stamp) 写锁尝试转换为读锁，失败返回 0，成功返回 state。

sleep sleep 的实现很简单，只需要记录开始时间，计算出结束时间，然后循环检查当前时间是否超过结束时间即可。使用这样的方法是为了避免使用 await。

join 因为主线程不能够阻塞，所以我们目前只实现了子线程之间的 join。每一个线程在结束时，都会给主线程发消息，主线程会标记该线程已经结束。当一个线程去 join 另一个线程时，就会向主线程发送 join 命令，并且带上想要 join 的线程的 id，主线程检查该线程是否结束，如果结束就让线程继续执行，否则等待。

使用实例如下：

```
class Scope {
    public static Thread t1;
    public static Object lock = new Object();
}
// 在另一个线程中调用:
Scope.t1.join();
```

2.1.4 消息优化

buildProxy 函数用于创建代理对象，该代理对象可以对目标对象进行属性的拦截和处理。在这个实现中，代理对象主要用于访问类的属性，并提供了优化功能。优化模式 buildProxy 部分代码如下：

```
//initWorker.js
//map存储所有改变了的key value
let changedObjects = new Map();
//map存储所有的主线程中的key value
let mainObject = new Map();
let buildProxy = () => {
    return new Proxy(target, {
        get: function (_target, propKey) {
            let key = className + '.' + propKey;
            // 如果是数组，递归创建代理
            if (Array.isArray(_target[propKey])) {
                return buildProxy(_target[propKey], key);
            }
            if (mainObject.has(key)) {
                return mainObject.get(key);
            }
            return _target[propKey];
        },
        set: function (_target, propKey, newValue) {
            ...
            _target[propKey] = newValue;
            mainObject.set(key, newValue);
            changedObjects.set(key, newValue);
            return true;
        }
    });
}
```

优化思路 根据 Java 的内存模型，程序在运行时会把数据从主存中复制到 Cpu Cache 中，从 Cache 中存取数据要比从主存中直接拿取要快的多。以此作为我们的优化思路，就是在子线程中暂存主线程中的共享数据，从而达到减少通讯次数和提高性能的目的。

但是这样造成的新问题就是，主存和 Cache 的数据一致性问题。为了解决这一问题，我们也参考了 JMM 中的 happens-before 原则，也就是

- 解锁操作发生在加锁操作之前
- 对 volatile 变量的写发生在后续所有的读之前
- 一个线程内的前一条指令发生在后一条指令之前

在我们的实现中，也尽可能地遵循这些原则，比如对 volatile 的写操作，会同步更新到主线程中，对 volatile 的读操作，会直接向主线程中获取，从而模拟 volatile 变量在 Java 中的行为。接下来讲述具体的实现：

为了保证数据的同步，我们在 get 时需要从主线程获取最新值，set 时更新主线程中的值。然而这样每次 get、set 都要发送消息，为此我们进行优化以减少消息数量，我们在进入同步代码块后即 Comm.sync 函数中调用如中 batch_qurey 函数向主线程发送一次消息获得主线程中记录的所有数据，之后 get 时不需要向主线程发送消息，在 set 时将修改的数据记录到 Map 中，并在退出同步代码块前即 Comm.unsync 函数中调用更新数据部分代码中 batch_update 函数将 Map 中数据通过一次消息更新到主线程中。这样，我们可以让消息的数量减小到两次。请求数据部分代码如下：

```
//initWorker.js
static batch_query() {
    const buffer = new SharedArrayBuffer(BUF_SIZE * 2);
    const arr = new Int32Array(buffer);
    let _objects = this.synchronizePostMessageWithData(arr);
    mainObject = new Map([...mainObject, ..._objects]);
}
static synchronizePostMessageWithData(arr) { // Int32Array
    postMessage({ 'command': 'batch_query', 'arr': arr });
    Atomics.wait(arr, 0, 0);
    let str = '';
    str += deserialize2MapStr(arr);
    let flag = Atomics.load(arr, 0);
    while (flag !== -1) {
        // 如果还有后续的数据传送，要继续接收
    }
    let objects = deserialize2Map(str);
    return objects;
}
```

更新数据部分代码如下：

```
//initWorker.js
static batch_update(changedObjs) {
    if (!changedObjs || changedObjs.size === 0) return;
    this.synchronizePostMessage({ 'command': 'batch_update', 'obj': changedObjs
        });
    changedObjs.clear();
}
static synchronizePostMessage(message) {
    const lock = createLock();
    postMessage({ ...message, lock });
    Atomics.wait(lock, 0, 0);
}
```

数据传输 接下来，我们以优化模式为例具体说说数据在主线程和 worker 之间的传递是如何实现的。

为了从主线程获得数据，如请求数据部分代码所示我们创建一个共享内存并将其发送给主线程，利用 Atomic.wait 实现阻塞。主线程收到消息后将存放数据的 Map 进行 JSON 序列化为字

符串，一次（共享内存足够大）或多次（共享内存不够大）写入到共享内存中并 Atomic.notify 唤醒 worker。相应的 worker 一次或多次读取数据后进行 JSON 反序列化将字符串转换为 Map。

为了将更改后的数据更新到主线程中，如更新数据部分代码所示我们将 Map 作为消息的内容的一部分，发送消息后阻塞。主线程收到消息后获取 Map 来更新数据，更新结束后唤醒 worker。

需要说明的一点是，从主线程获取数据不能和更新数据到主线程一样利用消息发送数据，是因为发送请求数据的消息后 worker 需要阻塞，阻塞期间 worker 无法接收到主线程发送的消息。所以我们利用共享内存，并且解决了共享内存的大小限制，即共享内存大小不足时，主线程将 JSON 序列化后的字符串分多次填入共享内存中，并设置第一个字节为标志位，-1 表示没有后续数据，0 表示还有后续数据，worker 中据此也进行相应处理。回复数据部分代码逻辑如下：

```
//WebWorker.js
function responseWithData(arr) {
    const JSONStr = JSON.stringify(Array.from(objects));
    const size = JSONStr.length;
    const batch = Math.ceil(size / (BUF_SIZE - 4));
    for (let i = 0; i < batch; i++) {
        //序列化
        populateArray(arr, JSONStr, i * (BUF_SIZE - 4), BUF_SIZE - 4);
        if (i === batch - 1) {
            Atomics.store(arr, 0, -1);
            Atomics.notify(arr, 0);
        } else {
            Atomics.store(arr, 0, size);
            Atomics.notify(arr, 0);
            while (Atomics.load(arr, 0) !== 0) { }
        }
    }
}
```

2.2 编译器拓展

原始的 JSweet 编译器并不支持 Java 多线程功能的识别与转译。针对这一问题，我们对其进行了拓展，使得 JSweet 能够识别 Java 中与并发相关的功能，并将其翻译成可执行的 TypeScript 文件。工具现支持对并发类、synchronized 关键字和 volatile 关键字的识别功能。

2.2.1 并发类处理

在我们的编译器拓展工作中，我们首先引入了对并发类的识别支持。这一支持主要涉及三个方面的工作：在 Util.java 类中新增对并行类的识别功能、新建 CvsAnalyzer.java 类以用于分析与存储可能用于不同线程通信的共享变量、更新 Java2TypeScriptTranslator.java 类使其可以完成变法类和方法的翻译工作。下面将详细介绍每个方面的改进内容及其实现细节。

Util 类更新。在 Util.java 类中，我们新增了一个方法 is_parallel(ClassTree clzTree)，该方法用于判断一个类是否具有并发特性。具体而言，我们将一个类视为并发类的条件是：该类继承自 Thread 类或者实现了 Runnable 接口。在实现过程中，我们使用了语法树的相关方法来获取类的继承和实现关系，进而判断该类是否满足并发类的条件。该功能为后续的多线程分析和处理提供了基础支持。

新建 CvsAnalyzer 类。为了进一步完善对多线程相关变量的分析和识别，我们新建了一个名为 CvsAnalyzer.java 的类。该类是我们编译器中的一个重要组件，用于分析 Java 源码中的交叉变量引用。这些交叉变量包括静态变量和 volatile 变量，它们可能在多线程环境下被使用，因此需要被准确地识别和处理。在实现过程中，我们采用了语法树遍历和分析的方式，识别并记录了每个类中的可能涉及多线程的变量。这些变量被存储在 capture_cvs 列表中，以用于后续的线程间通信。

在 CvsAnalyzer 类的实现中，我们采用了栈数据结构来管理类的作用域。通过动态进入和退出作用域，我们能够在遍历语法树时有效地跟踪变量的引用关系。其核心可分为作用域管理和交叉变量分析两部分。作用域管理部分有两个关键变量，分别为 clzScope 和 currentOutScope，用于记录当前类的作用域及其相关的外部类。交叉变量分析也有两个关键变量，分别为 cvsScope 和 volatileCvsScope，用于存储类与其交叉变量和 volatile 变量的映射关系。

此外，我们重写了 visitClass、visitMethod、visitVariable、visitIdentifier 和 visitMemberSelect 方法，使得我们能够按照我们特定的需求来分析每个类的结构，并准确地记录其中的静态变量和 volatile 变量。在识别变量时，我们还考虑了 Java 中的关键字问题，避免将关键字误认为变量引用。同时，为了提高识别的精度，我们还会将静态变量的引用以及外部类的成员变量纳入分析范围，以确保不会遗漏线程类中所引用的非线程类的静态变量。这样一来，我们就能够更全面地捕获类中的交叉变量引用，为后续的处理提供了准确的数据基础。

Java2TypeScriptTranslator 类更新。在 Java2TypeScriptTranslator.java 类中，我们拓展了关于类和方法的翻译部分，以便在识别到多线程相关的类时进行相应处理。具体而言，当识别到一个类是并发类时，我们会在翻译过程中添加交叉变量和 volatile 变量，以便在生成的 TypeScript 代码中进行不同线程间的消息传递和处理。

2.2.2 synchronized

synchronized 是 Java 中的一个关键字，在多线程共同操作共享资源的情况下，可以保证在同一时刻只有一个线程可以对共享资源进行操作，从而实现共享资源的线程安全。synchronized 是一种可重入、非公平、独占式的悲观锁。使用 synchronized 构造的锁主要分为对象锁和类锁。

对象锁包括同步代码块锁 (自己指定锁对象) 和方法锁 (默认锁对象为 this, 当前实例对象)。对于同步代码块锁的处理方式和加锁解锁的方法，我们在前面已经有提到。我们会用自定义的 Comm.sync() 指令和 Comm.unsync() 指令来进行加锁和退出锁，来完成对代码块的锁的实现，可以参考前面的 Figure 1 和 Figure 2 来看具体的表现。我们会在发现 synchronized 的地方进行加锁，但是需要注意使用指令退出锁的地方。在代码块的末尾，循环的跳出，return 跳出函数时，均需要考虑到是否已经跳出了代码块来进行解锁，以避免出现代码块锁死的情况。对于方法锁的形式，一般是通过 synchronized 修饰普通方法来实现的，我们先判断函数的修饰词中是否有 synchronized 存在，这里可以理解为将这个函数的整个代码块进行加锁。如前面所说，类似于 Java 的实现方式，我们使用 Comm.sync(this) 来进行加锁，由于是对整个函数加锁，这里我们只考虑了在代码块的末尾和 return 前用 Comm.unsync(this) 解锁。在处理相同的锁时需要依赖于每个锁对象的 key，这里的 this 并没有 key，我们通过在运行时用构造函数来对 this 的 key 进行随机数赋值，这样能满足后续分析的需求也能保证每个 this 的独特性。

类锁指 synchronized 修饰静态的方法或指定锁对象为 Class 对象。对于 synchronized 修饰静态的方法，我们判断函数的修饰词中是否同时有 static 和 synchronized 存在，然后考虑用指定锁对象为 Class 对象的方法来同时进行实现。Java 中 synchronized 的实现方式为类名 +.class。这

```java
class TestThread extends Thread {
    @Override
    public void run() {
        int i = 1;
        while (!Data.stop) {
            i++;
        }
        System.out.println("Thread1 stop: i=" + i);
    }
}

class TestThread2 extends Thread {
    @Override
    public void run() {
        Data.stop = true;
        System.out.println("Thread2 update stop to: " +
             Data.stop);
    }
}
```

(a) Volatile 可见性测试核心代码

```java
public class VolatileFunc implements Runnable {

    @Override
    public void run() {
        addnum();
    }

    public void addnum() {
        int i = 0;
        System.out.println("begin");
        while(i != 1000){
            mData.m = mData.m + 1;
            i = i + 1;
        }
        System.out.println(mData.m);
        System.out.println("end");
    }

    public static void main(String[] args) {
        Runnable Add = new VolatileFunc();
        Thread t1 = new Thread(Add);
        Thread t2 = new Thread(Add);
        t1.start();
        t2.start();
    }
}
```

(b) Volatile 原子性测试核心代码

里我们用 Comm.sync() 来实现时括号里面也用这种方式来表现。前面我们已经判断了并行类，所以我们会在编译时在里面加入一个 class 变量，同时会对它设置一个 key 并赋予一个初始值，将初始值设置为类名，这样在实现的时候就会根据类名考虑到不同的类来进行加锁解锁。这里也会像前面的实现来进行加锁解锁，保证所有线程需要的锁都是同一把。

2.2.3 volatile

Volatile 变量与 synchronized 关键字的区别在于，volatile 变量提供的是一种轻量级的同步机制，仅保证可见性和禁止指令重排序，而不提供原子性和互斥性。在我们的实现中，为了支持 volatile 变量，我们将 Java 中使用到的 volatile 变量存储至相应的 TypeScript 类中的 ___captured_volatile_cvs 字典中。此字典用于存储所有 volatile 变量及其对应的值。下面以一个示例来说明我们对 volatile 变量的支持。

示例 Figure 11a 说明了 volatile 变量的使用场景。在该示例中，如果 Data.stop 变量不使用 volatile 修饰符，则 testThread 会从自己的工作缓存中获取该变量的值，导致无法感知到其他线程对该变量的修改，从而无法正常退出循环。而若使用 volatile 变量，则 testThread 会直接从主存中获取 testThread2 更新后的值从而按预期退出循环。在 JavaScript 中，由于无法直接访问主内存，我们需要通过立即更新数据中心来确保 volatile 变量的可见性，保证多线程环境下的正确执行。而示例Figure 11b则是用于验证 Volatile 变量与 Synchronized 关键词的区别。由于 Volatile 变量不具备原子性，所以多个线程同时执行 mData.m = mData.m + 1 操作时，会出现值不稳定的情况，最后输出的结果会介于 1000 至 2000 之间。我们在转译的过程中保留 Volatile 的该项特点。生成的 JavaScript 脚本的执行结果总是一个介于 1000 至 2000 之间的不确定值。

在实际操作中，对于每一个多线程对象，我们会为其生成一个代理。当多线程对象访问某个变量时，其相应的代理对象会自动判断该变量是否为 volatile 变量。如果是 volatile 变量，则代理

Table 1: 消息数统计结果

	优化前					优化后					前平均	后平均
Actor	183	212	184	183	183	69	83	70	69	69	189	72
Alter	60	57	60	57	60	169	169	170	169	170	169.4	58.8
Array	1866	1563	1563	1565	1865	20	20	20	20	20	1684.4	20
BankAccount	185	182	181	181	182	75	75	75	75	75	182.2	75
Bicycle	378	378	378	378	378	141	138	141	141	141	378	140.4
CookAndCustomer	576	295	295	575	295	108	108	108	229	222	407.2	155
CookAndCustomerWithSleep	358	356	356	357	357	150	150	150	150	150	356.8	150
CookAndCustomers	3482	3426	3425	3411	3425	1333	1301	1301	1308	1301	3433.8	1308.8
Join	26	25	25	33	25	11	11	11	14	11	26.8	11.6
MultiProducer	603	603	502	581	591	246	252	248	248	246	576	248
NoSyncClass	40008	40008	40008	40008	40008	2	2	2	2	2	40008	2
NoSyncFunc	4012	4012	4012	4012	4012	10	10	10	10	10	4012	10
OnlySyncFunc	40012	40012	40012	40012	40012	10	10	10	10	10	40012	10
ParkingLot	311	311	311	311	311	107	107	107	107	107	311	107
ProducerAndAssmbler	466	463	325	442	463	177	177	169	169	119	431.8	162.2
ProducerAndComsumer	29030	29017	29027	29021	29017	10004	10004	10002	10006	10004	29022.4	10004
ReadWriteLock	-	-	-	-	-	870	873	870	870	870	-	870.6
StamptedLock	-	-	-	-	-	72	96	72	75	75	-	78
SyncClass	40010	40010	40010	40010	40010	10	10	10	10	10	40010	10
SyncFunc	40012	40012	40012	40012	40012	10	10	10	10	10	40012	10
TestRunnable	1	1	1	1	1	1	1	1	1	1	1	1
VolatileFunc	4006	4006	4006	4006	4006	4004	4004	4004	4004	4004	4006	4004
VolatileTest	1320	3959	3301	3672	3141	3669	3683	3322	2876	3457	3078.6	3401.4

对象会执行相应的 get 和 set 操作。对于 get 操作，代理对象会立即从数据中心读取该 volatile 变量的最新值，并返回给线程。而对于 set 操作，代理对象会立即将新值写入至数据中心，确保其他线程能够马上看到这个更新。这样做可以保证 volatile 变量的可见性，确保多线程环境下的数据同步和一致性。

3 消息优化评估结果

我们为了验证对 JSweet407 的消息优化结果，对项目 benchmark 中每个测试文件进行了十次实验，其中五次为优化前，五次为优化后的代码。Table 1记录了项目 benchmark 中所有测试运行完成之后所传递的消息包数量。从数据总体表现来看，我们基于 Java Memory Model 的优化方案效果显著，对大部分正确加锁的程序都有良好效果，

优化效果最好的是 NoSyncClass 用例，从 40008 个消息优化到 2 个消息传递。原因是这个例子中对某个共享变量在加锁后进行了上万次访问，而加锁操作只有两次，因此优化后的代码只会进行两次数据同步和两次消息传递。我们的优化并没有对用例 VolatileFunc 和 VolatileTest 起作用，因为这两个程序中都使用了 volatile 关键字，以这个关键字修饰的共享变量会在每一次写操作时更新数据到主线程，因此导致优化前后并没有明显的变化。还有两个例子 ReadWriteLock 和 StamptedLock 在优化前存在 bug，因此优化前数据为空。

```
class F {
    public int i;
    public static int j;
}
public class A {
    public static void main(String[] args) throws InterruptedException {
        T t1 = new T();
        T t2 = new T();
        F f = new F();
        t1.f = f;
        t2.f = f;
        System.out.println(f.i);

    }
}
class T extends Thread {
    public F f;
    public int j;
    public void run() {
        while (j < 5) {
            synchronized(f) {
                f.i++;
                System.out.println(f.i);
            }
            j++;
        }
    }
}
```

Figure 12: 跨线程共享变量

4 待支持的特性

我们在 JSweet407 的实现上做了一些简化处理，暂时还无法用于真实的多线程 Java 翻译。本章节会详细描述简化特性以及相应的后续解决思路。

4.1 完备的跨线程变量

如Figure 12所示, JSweet407 当前只支持静态跨线程变量可以正确读写 F 中的 public static int j。然而在 main 函数中，两个线程类 t1 和 t2 中的 f 成员变量全部指向同一个对象，因此在 t1 和 t2 中可以直接对成员变量进行直接读写操作，JSweet407 目前并未对这种情况进行处理。解决思路可以参照跨线程类的 proxy 方式，设计并发类的 proxy，发生读写操作时对子线程发送相应的消息。

4.2 主线程逻辑代码和数据处理、调度代码分离

JSweet407 现在的运行时实现中，主线程不仅需要运行 Java 源码主线程部分的代码逻辑，还需要承担数据中心和子线程调度加锁等任务。其中数据中心和子线程调度加锁等任务是通过消息接收事件触发的，因此会与主线程的代码逻辑产生冲突。例如主线程在处理一个极其耗时的任务（while(true)）时，是无法响应事件循环中的消息请求的。此时子线程的通信将会全部阻塞直到耗时任务结束，主线程才会处理事件循环中消息队列。

另一方面，JS 的主线程环境无法进行阻塞操作，因此所有的线程协作机制将难以实施。例如主线程中 join 一个子线程，此时从编译的角度来说，主线程有两个选择：1. 在 join 这个点直接阻塞 2. 把后续的代码转成异步形式，在 join 成功之后回调。显然选择一更加简单也更符合 Java 原本的语义，然而受限于 JS 不可阻塞的主线程机制，无法直接支持这种行为。

以上两个问题挑战的相应的解决思路比较简单，将原本 Java 主线程的逻辑代码全部编译成子线程，主线程只承担数据的通信与加锁调度任务。分离之后主线程不会有耗时的计算任务，并且 join 的代码此时在子线程中，可以方便地执行阻塞操作。

4.3 并行对象 public 方法调用

JSweet407 暂时不支持子线程的 public 方法调用，因为方法调用涉及到子线程与主线程之间的数据传输问题。如前文所述，暂时并不支持主线程与子线程之间的数据交换问题。