有道云链接：http://note.youdao.com/noteshare?id=ffe880a08cfbad4ec7017cd857d753c5&sub=0ADD4EC62D434EF6BE7A6F195507D1A6

STW，即Stop The World。

为什么需要STW呢？试想你妈给你打扫房间的场景：把你撵出去，关上门，打扫干净，打开门，数落你，揍你...一套标准化流程后，房间干净了。打完你，你妈的心情变好了，打麻将都能多赢点。这里面有个关键环节：把你撵出去。尽管在打扫方面的过程中你可能不会制造垃圾，但是你的存在就有这个风险，所以必须把你撵出去。这话不是我说的，是从你妈的行为中揣摩出来的。^_^

试想，如果不把你撵出去，你妈打扫垃圾的同时，你又陆陆续续制造了垃圾，那这场打扫房间的行动是不是变成了无法结束的行动啊。或者到某个时间点，你妈打扫了一半走了，丢下一句话：朽木不可雕也，孺子不可教也。

垃圾收集器也是一样的，为了保证清理垃圾的完整性，在某些环节，就会STW。比如所有垃圾收集器中都有的一个阶段：初始阶段，即扫描根对象，需要STW。小伙伴们看过的几乎所有资料，讲到这基本就没了对吧。但这不是子牙老师我的风格，咱们接着往后面说。

STW

JVM中要做到STW是很难的。为什么这么说呢？因为需要考虑很多很多因素。

一、JVM中存在多种类型的会发生改变内存行为的线程：

1. 执行业务逻辑的用户线程
2. 执行native方法的Java线程
1. 执行垃圾收集的GC线程（并行并发垃圾收集需要考虑）
2. 执行即时编译的JIT线程

二、每种类型的线程个数，在需要STW的那一刻，可能都不止一个。

三、每种类型的线程，在需要STW的那一刻，执行到的代码位置也未可知。
四、每种类型的线程阻塞的点还不能随机。因为线程在阻塞前需要更新OopMap。OopMap是什么？你可以理解成是记录这个线程一路跑下来经历过的所有Java对象的集合。为什么要有

OopMap呢？因为没有的话，你就得扫描整个栈，去查找根对象。这里说的只是查找根对象的一种情况哈，勿抬杆，我会记仇。^_^

如何暂停线程

听我这么一分析，好像确实很复杂哈。那如果是你来实现，你会怎么解决呢？小伙伴们可以想一想。经常想这样有深度的问题，有利于提高你的思考深度。我们还是来看看JVM是如何高明地解决的吧。

如果线程随便哪个位置阻塞都合适，这个问题就会简单一百倍。但是这里简单了，给其他地方就带来了灾难。就是说线程阻塞前需要更新OopMap，如果不更新，没有这个数据的话，GC时就需要扫描所有线程的所有栈的所有栈帧来查找根对象。OopMap的存在，其实又是一种空间换时间的策略。因为相比内存的价格，降低GC延时明显更重要。

但是JVM的执行流那么多，何时？在什么地方？更新OopMap呢？这就是安全点存在的意义。安全点同时解决了STW及更新OopMap。其实也可以这样说，不理解安全点就无法理解STW，甚至于无法理解GC。

安全点

安全点涉及的知识点非常多、非常底层。本篇文章就讲安全点中与STW相关的知识点。其他的知识点后面会写系列文章展开讲。感兴趣的小伙伴可以关注我公众号关注我的发文动态：硬核子牙。

这段代码是大家看GC源码时经常看到的
SafepointSynchronize::begin
我把hotspot源码中核心的代码粘过来

```
// Roll all threads forward to a safepoint and suspend them all
void SafepointSynchronize::begin() {
  ……
  // Make interpreter safepoint aware
  Interpreter::notice_safepoints();

  ……
    os::make_polling_page_unreadable();
  ……

  // wait until all threads are stopped
  while (_waiting_to_block > 0) {
    ……
```

这段代码到底做了哪些事情呢：

1. 告诉JVM马上要开始GC（下雨）了，开始做准备工作了（准备收衣服了）。本质就是修改一些属性位。比如第5行代码，通知解释器做好准备工作，迎接GC到来。
2. 将polling_page对应的物理页设置成不可读状态。**这步非常非常重要**。等下说。
1. 不停检测，确定是否所有的线程都已进入安全点。只有都已进入安全点，才能执行GC逻辑。

STW的真面目

安全点是如何解决让所有的线程都阻塞的呢？开启安全点为什么要将物理页的属性改为不可读呢？

因为JVM在生成执行流代码的时候，都会在适合作为安全点的地方插入一段代码

```
1 test   %eax, os::_polling_page
```

这段代码就是安全点的本质，也是触发STW的本质。什么意思呢？如果os::_polling_page对应的物理页属性是可读的，这段代码并没什么特殊意义。但是如果是不可读的，读的时候就会触发段异常，对应的操作系统信号：SIGSEGV。JVM捕获了这个异常，并进行了处理。所有的线程都是在这个地方STW的。

```
1  JVM_handle_linux_signal(int sig,
2                          siginfo_t* info,
3                          void* ucVoid,
4                          int abort_if_unrecognized) {
5  ……
6      // Check to see if we caught the safepoint code in the
7      // process of write protecting the memory serialization page.
8      // It write enables the page immediately after protecting it
9      // so we can just return to retry the write.
10     if ((sig == SIGSEGV) &&
11         os::is_memory_serialize_page(thread, (address) info->si_addr)) {
12       // Block current thread until the memory serialize page permission restored.
13       os::block_on_serialize_page_trap();
14       return true;
15     }
16 ……
```

这就是安全点难的地方，涉及到的知识点太多太底层！其实我搞手写JVM小班的核心目的不是带你写一个JVM，其一是让你通过手写JVM了解hotspot的体系，你才能看得懂hotspot源码。其二，也是最核心的，掌握底层。因为掌握了底层，你对技术就没有恐惧之心了，你会觉得你无所不能。事实上，相对的无所不能是可以做到的，只是需要时间沉淀。啰嗦了两句哈。

GC结束后唤醒所有阻塞的线程，小伙伴们应该能想到是在哪里？如何唤醒的了吧

```
// Wake up all threads, so they are ready to resume execution after the safepoint
// operation has been carried out
void SafepointSynchronize::end() {
    ……
    // Release threads lock, so threads can be created/destroyed again. It will also starts all threads
    // blocked in signal_thread_blocked
    Threads_lock->unlock();
    ……
```