Click to verify

執行緒安全是程式設計中的術語，指某個函式、函式庫在多執行緒環境中被呼叫時，能夠正確地處理多個執行緒之間的公用變數，使程式功能正確完成。 假設有間銀行只有 1000 元，而兩個人同時提領 1000 元時就可能會拿到總計 2000 元的金額。為了避免這個問題，該間銀行提款時應該使用互斥鎖，即意味著針對同一個資源處理時，前一個人提領交易完成後才處理下一筆交易。 但這種手法會使得效能降低。 一般來說，執行緒安全的函式應該為每個呼叫它的執行緒分配專門的空間，來儲存需要單獨儲存的狀態（如果需要的話），不依賴於「執行緒慣性」，把多個執行緒共享的變數正確對待（如，通知編譯器該變數為「易失（volatile）」型，阻止其進行一些不恰當的最佳化），而且，執行緒安全的函式一般不應該修改全域對象。 很多C庫代碼（比如某些strtok的實現，它將「多次呼叫中需要保持不變的狀態」儲存在靜態變數中，導致不恰當的共享）不是執行緒安全的，在多執行緒環境中呼叫這些函式時，要進行特別的預防措施，或者尋找別的替代方案。 在下面這段代碼中，函式increment_counter是執行緒安全的，但不是可重入的。 #include int increment_counter () { static int counter = 0; static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; pthread_mutex_lock(&mutex); // only allow one thread to increment at a time ++counter; // store value before any other threads increment it further int result = counter; pthread_mutex_unlock(&mutex); return result; } 上面的代碼中，函式increment_counter可以在多個執行緒中被呼叫，因為有一個互斥鎖mutex來同步對共享變數counter的訪問。 但是如果這個函式用在可重入的中斷處理程式中，如果在pthread_mutex_lock(&mutex)和pthread_mutex_unlock(&mutex)之間產生另一個呼叫函式increment_counter的中斷，則會第二次執行此函式，此時由於mutex已被lock，函式會在pthread_mutex_lock(&mutex)處阻塞，並且由於mutex沒有機會被unlock，阻塞會永遠持續下去。 簡言之，問題在於 pthread 的 mutex 不可重入。 解決辦法是設定 PTHREAD_MUTEX_RECURSIVE 屬性。然而對於給出的問題而言，專門使用一個 mutex 來保護一次簡單的增量操作顯然過於昂貴，因此c++11中的原子變數提供了一個可使此函式既執行緒安全又可重入（而且還更簡潔）的替代方案： #include int increment_counter () { static std::atomic counter(0); // increment is guaranteed to be done atomically int result = ++counter; return result; } Thread-safe Tcl Extensions（頁面存檔備份，存於網際網路檔案館）（wiki page） Thread-safe design Article "Design for thread safety" by Bill Venners Article "Write thread-safe servlets" by Phillip Bridgham 取自「 著作: 蕭沖何謂thread-safe? 這個問題我看過許多論壇都有討論過，都總讓人覺得不很滿意。在此，筆者想要用更logical的方式來把議題說清楚。 首先，我們要了解它的定義! 定義若都不明白就難以判別安不安全了!thread是什麼呢? 可能也有人不太了解。就從這裡開始... 當cpu處理一段(區塊)的程式碼時，從開始的第一行程式碼來跑就算是thead的開始，直到區塊的最後一行程式結束，就算是thread的結束。所謂的「主thread」，在dos (或dos like)程式裡，就是指main( )這個函式的開始到結束的一個thread。而主thread以外的一個子thread則是指程式人員自行在主thread裡再定義一個「程式區塊」，並請cpu同步的去執行那個區塊。由上面的thread的定義來看，我們可以進而推論所謂的multithread這個字其實是有三種情形的，哪三種呢?1/ 相同的一個程式區塊，請cpu"分出多身"來執行它! (一段code被new或說create成數個thread)2/ 不同的程式區塊，請cpu各別分時的去執行每個區塊 (幾段code個別被new或說create成個別的thread)3/ 混合以上二種情形所以，當我們在討論多緒/多線程(multithread)的問題時，我們要先去了解問題是上面三類的哪一類。若是第一類，那就要注意若有使用全域變數，那就極有可能是不安全的。若是第二類，那也要考慮是否不同的thread code是否有共用某同一個全域變數。我們再把「程式區塊」給解析一下。它可以是一個function，或是一個class(裡面有成員函式)，或是上述二者的混合體。因為function或是class都會使用到變數，於是多緒時變數的共享就成了一個很大的問題。這就是為什麼會有thread-safe的這個議題了!到此為止，我們可以想像若一個程序(process / 主thread)裡，有大於等於一個子thread在跑，就會產生thread-safe的問題。以最簡單的情形來說，mainthread+user自定的一個thread。假設Button元件按下後會刪除memo元件裡某個String，而同時間的一個thread則會取用memo裡的String。假想當thread在取用memo中某String做分析的時候，user不知情的按下button，那麼thread是否有可能會取到不正確的String? 若不經過特別的安全設定，肯定「取用」時時而會出問題!因此，到底安全或不安全的根源就在那執行的區塊程式碼上了(以上例就是取用的過程)。我要補充一個重要的事，其實每一個thread都有自己的一個專屬stack，所以auto級(local)的變數是不會有共享的問題，因為它們都是存放在thread各自的記憶體空間內，彼此不甘擾。也就是說，若你的程式區塊中，所有使用到的變數都是local(auto)變數，那你這個區塊，或說這個thread，就算是thread-safe !! 相反的，若你的區塊中有用到全域的變數，那麼你的這個thread就有可能是不安全。請注意看，是有可能，並非一定是。如何把含有全域變數的thread分析出安全或不安全? 其實這是很直覺的。比如說某全域變數int g被數個thread使用到，但每個thread裡只引用到一次，那算安全嗎? 答案是...還算安全(註:極度的嚴僅來說不100%，但98%不為過)。因為單一基本變數的一次存取cpu處理極快，不太會有衝突。又，若是某全域結構體變數 struct tag 被引用存取到一次，那算安全嗎? 答案是...不太安全，因為一個結構體的存取可能涉及到多個微指令。進一步的用上面的例子來說，若int g被「程式區塊」(thread code)引用到2次(含)以上，那就肯定不安全了。比如說第一次是把g設成k，第二次把g再設給某變數i，若thread1正在處理第二次引用，而thread2在處理第一次引用。那thread1可能會得到意外的答案!(g值非它第一次設成的k值，而是變成thread2裡的k值)如何讓不安全的thread變成安全的?? 其實主要的方法有二類。1/ 存取共用全域變數時加上lock的機制，而這lock的機制在windows上可以使用mutex、criticalsection、event、semaphore等物件配合WaitForSingleObject或WaitForMultipleObject二個api來控制。其原理就是當某thread在取用資料時，其他的通通都停止並等待，直到該thread取用完。2/ 把存取全域變數的動作通通交給某一個thread全全處理。比如說你有thread1，thread2，thread3。當要存取全域的struct變數時，thread2就把工作交給thread1做，然後自己停住等thread1完成，thread3也把工作交給thread1做，並等待... 。這原理應該容易理解，因為不安全的工作全都交給"單一"的thread來處理時，那份工作就安全了呀! 當然，這也不禁讓我們覺得，這樣效能非常的差!通通都在等thread1。 不過還好的是，若少少的用還ok，若許多的工作都交給某一thread順序來處理，那就真的沒意義了! 而這種方式其實就是bcb或delphi裡的synchronize這個方法的處理方式! (請看最下面2011年我的補充說明)後註: class 其實也像是一種function，所以class本身也是存在著安全與否的問題。倘若該class裡的menthod有用到全域的變數時，該class也算是不安全的! 而許多的VCL就是這類的class!註2: 若什麼是auto變數、stack都還不清楚，請先自行google或查程式語言的書，這是很基本且重要但易被忽略的章節。 Page 2 Version Ships ('ed) with dll name 1.0 Windows 95/98/ME/NT riched32.dll 1.0 Exchange 4.0 for Windows 3.1/WFW richedit.dll 2.0 Office 97, Windows NT/98 riched20.dll 2.1 BiDi Office 97 riched20.dll 2.5 Windows CE, Pocket Word riched20.dll 3.0 Office 2000, Windows ME/2000/XP riched20.dll 1.0 emulator Office 2000, Windows 2000/XP/Vista riched32.dll 3.1 Windows Server 2003, Vista riched20.dll 3.5 Windows CE, eBooks ebriched.dll 4.0 Office XP riched20.dll 4.1 Windows XP SP1, Tablet, Vista msftedit.dll 5.0 Office 2003 riched20.dll 5.1 Windows CE, Pocket Word riched20.dll 6.0 Office 2007, Encarta Math Calculator riched20.dll RichEdit 1.0 Features Basic nonUnicode editing, cut/copy/paste, file streaming Basic set of character/paragraph formatting properties Message-based interface plus OLE interfaces: IRichEditOle and IRichEditOleCallback Vertical text and IME support (FE builds only). WYSIWYG editing using printer metrics Different builds for different scripts Common-control notifications plus some new ones Plain text and RTF files Pen-enabled and understood gestures for use with Pen Windows RichEdit 2.0 Additions Unicode internally + able to read/write using codepages International line breaking algorithm Find Up/Down. Magellan mouse support. Multilevel undo BiDi (RE 2.1) and FE support including level 2/3 IME, dual font, keyboard linking, smart font apply AutoURL recognition. Word UI Plain/rich, single-line/multiline, scalable architecture Password and accelerator control options Windowless interfaces (ITextHost/ITextServices) Better display (mixed fonts use off-screen bitmap), system selection colors, transparency support TOM (Text Object Model) dual interfaces Character formatting additions include background color, locale ID, underline type, superscript/subscript. Paragraph formatting additions include space before/after, line spacing. Roundtrip all Word Format Font/Para dialog properties Extensive code stabilization, testing, performance increase RichEdit 2.5 Additions First Windows CE version. Used by Pocket Word Outline view, normal and heading styles RTF additions Minor UI improvements Western languages only RichEdit 3.0 Additions Used for emulating RichEdit 1.0's Zoom Italics caret/cursor. URL hand cursor Paragraph numbering (alpha, numeric, Roman) Simple tables (no wrap in cells) More underline types, underline coloring, hidden text More of Word's default hot keys, e.g., accent dead keys, outline view, numbering Smart quotes (English only), soft hyphens Use Office's LineServices component to break/display lines. Used for complex scripts and options like center, right, decimal tabs, fully justified text Complex script support for BiDi, Indic, and Thai with help from LineServices and Uniscribe components Font Binding based on charset, which acts as writing system ID Codepage-specific stream in/out UTF-8 RTF. Used preferentially for cut/copy/paste. Can be streamed in/out. Office 9 IME support (MSIME98) including Reconversion, Document feed, Mouse Operation, and Caret position features AIMM component IME support for nonFE systems. Increased freeze and undo/redo control Font increment/decrement function System edit control, list box, and combo box controls Alt+x input method Used to emulate RichEdit 1.0's RichEdit 3.5 Additions Second Windows CE release. Used by eBooks Screen-size pagination Text wrap around objects flushed left/right Custom ClearType support Enhanced East Asian typography RichEdit 4.0 Additions Multilevel tables Autocorrect Improved autoURL detection Friendly name hyperlinks Font binding according to writing system (generalization of charset) Indic support Vertical text Support for the latest IMEs Speech and handwriting input (Windows Text Services Framework) More standard hot keys Many security fixes (3.0 has also) RichEdit 5.0 Additions Multiselection, smart drag&drop Better nested tables, horizontally merged cells Better font binding/international support More underline styles, small cap & shadow emulation Binary file format: "parsed XML" Partial XHTML reader/writer Subpixel ClearType support Better RTF handling, e.g., multilevel lists URL tooltips Many bug/minor-request fixes Improved ink support, especially for OneNote Advanced East Asian typography Initial PTS integration, including object tight wrap Infrastructure for math, ruby, warichu, tatenakayoko Text trackers and blobs RichEdit 5.1 Third Windows CE release. Used by Pocket Word Various UI and RTF enhancements RichEdit 6.0 Additions High-quality editing & display of math Formula autobuildup Create and support math linear format More list numbering options Simple "visi" mode URL improvements Multistory: high-perf cut/copy/paste, rich scratchpads, WP infrastructure Text Object Model 2 Display enhancements, e.g., word underline, horizontal scaling Table UI adds, e.g., column resizing OfficeArt/PowerPoint enhancements Overlapping lines, drop caps & other ePeriodicals improvements Device independent layout Virtualized OS: "hDC" is totally opaque Multiple columns Myriad security fixes Page 3 來自 title says it all: you can use an empty action attribute on a form to submit that form to the current page. This means you don't need to use server-side scripting (using REQUEST_URI or PHP_SELF or whatnot) to write the current URL into the HTML.The following is perfectly valid: Now beware, the action attribute is mandatory, and it must contain a valid URI. But according to the URI RFC, an empty URI is still a URI: 4.2. Same-document References A URI reference that does not contain a URI is a reference to the current document. In other words, an empty URI reference within a document is interpreted as a reference to the start of that document, and a reference containing only a fragment identifier is a reference to the identified fragment of that document. Traversal of such a reference should not result in an additional retrieval action. However, if the URI reference occurs in a context that is always intended to result in a new request, as in the case of HTML's FORM element, then an empty URI reference represents the base URI of the current document and should be replaced by that URI when transformed into a request. So there you have it. Enjoy. GSS 技術部落格 在這個園地裡我們將從技術、專案管理、客戶對談面和大家分享我們多年的經驗，希望大家不管是喜歡或是有意見，都可以回饋給我們，讓我們有機會和大家對話並一起成長！ 若有任何問題請來信：gss_crm@gss.com.tw Concurrency (並行) 和 Parallelism (平行) 是非常像的概念。前者是說不同的計算並行地執行，相較於序列化 (Sequentially) 執行，在一個計算結束之前另一個就開始了；後者則是讓同個計算工作同時分工一起執行。 乍看之下這兩者要講的概念其實很像，採用 The Art of Concurrency 的定義如下： A system is said to be concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously. The key concept and difference between these definitions is the phrase “in progress.” 一個系統稱做並行是當他可以同時執行兩個行進中的任務。一個系統稱為平行是當它可以同時執行多個任務。兩者關鍵差異是「行進中」。 並行化可以在單核處理器 (Single Core) 上可以藉由交錯地執行任務來辦到，有趣的是並行化也可以在多核處理器上以平行的方式執行。 某種程度上並行跟平行做的事情滿像的，但「行進中」精關點出兩者區別，並且英文中 simultaneously 有 “happening or being done at exactly the same time” 的特性。 對於平行和並行不熟的同學可以考慮複習 Operating System Concepts 第四章「Threads & Concurrency」，或是 Operating Systems: Three Easy Pieces 的「Concurrency」章節。 而在並行與平行中，為了確保程式執行邏輯正確性和資料正確性，就會需要討論 Reentrancy 和 Thread-safe 分別是啥玩意。 ¶ 2. Reentrancy 在電腦科學中，Reentrancy (可重入) 代表一個程式或子程式裡面的程式碼可以「在任意時刻發生中斷 (Interruption)，然後作業系統排程 (Schedule) 執行另外一段程式碼，當再次回來執行這段程式碼時卻不會出錯」。 至於為甚麼程式會甚麼會中斷，有可能是內部自己的行為像是 jump 或 call，抑或是外部行為像是 Interrupt 或 Signal。也就是說，在程式執行的過程中，中斷的發生無關乎作業系統 (OS) 是否存在，沒有 OS 情況下，自身的行為也可能會引發中斷，所以還是要注意 Reentrancy 影響。 Reentrancy 在單執行緒的形況下就可以做討論，像是程式是否能在被 OS Interrupt 後，再次直接恢復執行？換句話說，如果被中斷後要能繼續執行，應該要是 Reentrant Code，否則回來的時候答案就錯了。這邊還有個有趣的問題，那就是 Interrupt Handler 需要是 Reentrant 嗎？簡單的答案是，除非你的 Handler 是巢狀的 (一個接一個呼叫)，否則是不需要擔心，另外像是 Linux 會有遮罩阻止另一個 Interrupt 打斷目前的 Interrupt。 Reentrancy 非常重要是因為在 Concurrent Programming (並行程式設計) 開發中，我們需要確保異步程式 (Asynchronous Program) 在做任務切換的時候，中斷不會影響我們執行正確性。此外當我們使用遞迴的時候，預期他就是 Reentrant 否則會出錯。 ¶ 3. Thread-safe 另一方面，Thread-safe (執行緒安全) 是指某個函數、函數庫在多執行緒環境中被呼叫時，能夠正確地處理多個執行緒之間的公用變數 (全域變數、共享變數)，使程式功能正確完成。 Thread-safe 在平行程式中很重要，因為平行計算的過程中，往往很多資料是共用的，這樣處理資料的時候，很容易讓資料產生 Race Conditions，因此如何確保資料在每個 Thread 都有被正確進行讀寫就很關鍵。 所以 Thread-safe 基本上就是在避免資料發生 Data Racing，實現機制可以使用 Reentrancy，但也可以使用 Thread-Local Data (只存在自己執行緒的資料)、不可改變的物件 (Immutable Objects)、互斥鎖 (Mutex)、原子化操作 (Atomic Operations)。 ¶ 4. Reentrancy VS Thread-safe 所以重點來了，那 Reentrancy 與 Thread-safe 關係是甚麼呢？ 兩者其實互不等於對方，但卻有部分交集，亦即 Reentrancy 可以是也可以不是 Thread-safe，反之 Thread-safe 對 Reentrancy 也是如此。 以下就直接用程式碼範例來做不同情況下的解釋。 ¶ 4.1 Reentrancy | Thread-safe int t; void swap(int *x, int *y) { t = *x; *x = *y; // 這邊可能呼叫 my_func(); *y = t; } void my_func() { int x = 1, y = 2; swap(&x, &y); } Reentrancy t 放在外部，如果 swap 執行到一半，接著另一個人修改了 t，那回來繼續執行時，行為就不正確。 Thread-safe t 是 Global 當其他 Thread 呼叫 my_func 時，t 可能剛好屬於同個 Context (程式執行環境)，則 t 的行為就會難以預測。 ¶ 4.2 Reentrancy | Thread-safe #include // `t` 是每個 thread 自己的 thread_local int t; void swap(int *x, int *y) { t = *x; *x = *y; // 這邊可能呼叫 my_func(); *y = t; } void my_func() { int x = 1, y = 2; swap(&x, &y); } Reentrancy 雖然 t 是 Thread 自己的，但同個 Thread 中可能在多次呼叫中改變 t。 Thread-safe t 現在是 Thread 自己的，別的 Thread 不可能去影響 t。 ¶ 4.3 Reentrancy | Thread-safe 以下為刻意製造的情境，但可以想成程式很複雜的情況下會有的狀況。 int t; void swap(int *x, int *y) { int s; // 存下全域變數 s = t; t = *x; *x = *y; // `my_func()` 可以在這邊被呼叫 *y = t; // 恢復全域變數 t = s; } void my_func() { int x = 1, y = 2; swap(&x, &y); } Reentrancy swap 執行前後 t 都會保持不變。注意這邊關鍵其實是因為 swap 不會對外部造成影響，亦即所有變數變化只保留在 swap 裡面。 Thread-safe ¶ 4.4 Reentrancy | Thread-safe 這個例子中的解決方法意外的容易，把全域變數拿掉就沒事了。 void swap(int *x, int *y) { int t = *x; *x = *y; // `my_func()` 執行 *y = t; } void my_func() { int x = 1, y = 2; swap(&x, &y); } Reentrancy Thread-safe ¶ 5. Reentrancy 與 Thread-safe 的原則 看過上面的範例之後，所以我們如果要寫出 Reentrancy 或 Thread-safe 可以遵守以下原則。 Reentrancy： 不能含有 static (global) 非常量 (non-constant) 數據。 不能返回 static (global) 非常量數據的地址。 只能處理由呼叫者 (Caller) 提供的數據。(透過參數傳入) 呼叫的函數也必需也是 Reentrant。 Thread-safe： 基本上只要避免 Race Condition 就好 Lock 是你的好朋友 ¶ 6. Reentrant 與 Thread-safe 函式庫 Reentrant 與 Thread-safe 函式庫在我們寫平行程式或開發異步程式的時候很重要。 在 GNU C 函式庫中，有分以下安全 (Safety) 等級：MT-Safe (Multi-Thread-Safe)、AS-Safe (Async-Signal-Safe)、AC-Safe (Async-Cancel-Safe) 以及各種不安全的等級。 一些 C 標準函式庫的函式，像是 ctime 和 strtok 就不是 Reentrant。但通常他們會有一個對應的 Reentrant 版本，名稱通常會是 _r 後綴，例如 strtok_r 或 rand_r。 事實上我們也可以透過 man 命令 (Command) 來確認，例如在 Ubuntu 16 上我查 man rand_r，得到以下結果 (片段)： ATTRIBUTES For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value
rand(), rand_r(), srand()	Thread safety	MT-Safe

可以看到我們查到 rand_r 是 MT-Safe，意味著我們可以用他在平行程式中。 MT-Safe 強調在多執行緒 (MT) 的環境中，原本預期的功能 (如這邊的亂數產生器) 仍可正確地展現，且不該存在因為放到多執行緒而有的錯誤 (即 functional safety)。 所以要注意的是，MT-Safe 不代表程式碼就是完全安全的，例如當連續呼叫 MT-Safe 的程式的時候，行為可能會是無法預期的。 那你可能就要問了，那如果我在平行程式上用非 MT-Safe 的函數呢？那可能會有兩種可能，一種是你得到錯的答案，因為他就不是安全的，第二種是你可能效率會比較差，因為他可能會不斷去搶外部變數。舉例來說，假設你在平行程式去呼叫 rand 而非 rand_r，那你將會發現亂數產生的速度會極慢 (甚至會不正確)，因為 rand 的實作中，就有使用 static。 ¶ 7. 參考資料 警告：我不是一个多线程编程的专家。事实上，甚至不能说我可以胜任这个东西。我的整个职业生涯，需要用到多线程来实现代码的次数也不多。所以请用一个怀疑的态度来看待我以下的所有观点。 有一个我经常问的问题是：“这个代码是线程安全的吗？”，要回答这个问题，我们很显然要先知道线程安全是什么。 但是在我们讨论这个问题前，我想先把另外一个问题搞清楚，还有一个我稍微问的没有那么多的问题是“Eric，为什么Michelle pfeiffer在照片里总是这么好看”，要回答这个难题，我咨询了一下维基百科： 一个上镜的对象指的是对象总能在照片里面很有吸引力或引人注目 为什么Michelle pfeiffer在照片里那么好看？因为她很上镜。 显然的。 好的，我很高兴我们先把这个谜团解决了，但是不知怎么的又有另一个难题现在悬在我的手中。维基百科是这么定义线程安全（thread safe）的： “A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads.” 代码的一部分如果能在多线程同时执行时运行正确那么就是线程安全的 就跟上镜一样，这很明显，当我们问“这个代码是线程安全的吗？”我们实际上是在问“这份代码在特定情况下可以运行正常吗？”那么我们要如何来定义代码是不是正常呢？这个我们还没有任何定义。 维基百科写到： “In particular, it must satisfy the need for multiple threads to access the same shared data, …” 尤其重要的是，必须要满足多线程访问共享数据的的需求 这好像是对的。这好像就是人们想要表达线程安全的意思。 但是： “…and the need for a shared piece of data to be accessed by only one thread at any given time.” 还有...被需要共享的数据同一个时间只能有一个线程来访问 到现在我们才开始讨论到写线程安全代码的技巧，而不再是讨论线程安全到底是什么。 把一个数据锁住，这个同一个时间只有一个线程进行访问只是实现线程安全的其中一个技巧，而不是线程安全的定义。 我不是说这个定义是错的。相较于之前的线程安全的定义，这个定义完全不差。我实际上想说的是，这个定义很含糊，跟之前的“在某些情况下执行正确”表述没有什么不一样。因为，当我问“这个代码是线程安全的吗？”我常常要往回退一步讲“你是在讨论哪个场景下的应用？”和“到底是在哪个场景下表现出哪种结果才算是正确的行为？” 人们在谈论线程安全的时候，总会产生很多不一样的问题。 比如，当我告诉你我有一个“线程安全的可变队列”，你可以把这个用在你的程序中吗。你很高兴的写了一个线程来执行它然后又用另外一个线程频繁的来新增或者移除可变队列中的对象： if (!queue.IsEmpty) Console.WriteLine(queue.Peek()); 然后你的代码在peek抛出QueueEmptyException时崩溃了，到底发生了什么？我说这个是线程安全的，但是你的代码却在一个多线程使用场景中崩溃了。 当我说“这个队列是线程安全的”的时候，我的意思是，这个队列会持续不断的维护其内部的状态，无论其他的线程以什么样的顺序来单独执行。但是我没有说在一个队列顺序周期内多次操作它还会自行维护状态。简单的说，我所理解的正确行为跟你所理解的正确行为不是一回事。我只关心它会不会崩溃，但是你在乎的是队列能不能正确的返回相应的方法。 在这个例子中，你和我可能都说到了不同的线程安全。可变数据的线程安全通常指的是要确保运行中共享数据永远都是最新的状态，即使是我们操作的过程中逻辑并不连贯，就跟上面的例子一样。不可变数据结构的线程安全通常讲的是执行过程中逻辑是一致的,尽管你要找的不可变快照是过时的数据。 现在的问题是要不要获取第一个元素取决于“过时”的数据，要设计一个在所有场景下都不允许有“过时”数据的真线程安全可变数据结构是十分困难的。 如果你真的要将“peek”操作做到真正的线程安全，你需要一个新的方法： if (!queue.Peek(out first)) Console.WriteLine(first); 这个是线程安全的吗？看起来的确好多了。但是如果执行完peek之后，另一个线程执行了出队(dequeues)操作呢？代码不会崩溃，但是你已经完全把之前程序的行为都改变了。在之前的逻辑里，如果测试后有另一个线程对其进行了“出队”操作，那么第一个元素会是什么，要么程序崩溃，要么就将第一个元素的最新状态打印出来。 现在你打印的第一个数据是过时的。那么这是正常的吗？如果我们不总是想要对最新的数据进行操作的话。 这里稍微暂停一下，实际上，之前版本的代码也有这个问题。如果你在进行peek成功操作之后有另一个线程执行了出队操作，而且是在你打印操作之前，那么你可能打印出的就是一个过时数据。 如果你是想要无论何时都打印出最新的数据呢？你实际上想要实现的线程安全操作是：queue.DoSomethingToHead(first=>{Console.WriteLine(first);}); 现在这个队列的作者和使用者都在怎么使用它上达成了一致意见，所以这才是真正的线程安全，对吗？ 除非...有超级复杂的需求。如果在某个需求里触发了代码执行另外一个线程，从而导致了队列操作，进而产生了死锁。那么死锁算不算一个正确的行为，如果不是，那么这个方法是真正的安全吗？ 是的。 现在你应该理解了我的观点，就像我之前所说的那样，如果不沟通好具体是哪种安全威胁或者没有经过验证的话，说要编写一个安全的代码是完全没有帮助的。 同样的,如果不讨论哪种不合适的行为会利用线程安全机制，哪种行为预防不了而去说这个代码是线程安全的也是完全没有帮助的。 线程安全和代码规范大同小异。 你同意以某种形式去和一个对象进行沟通，如果你符合规范的话，它就会给你返回一个正确的结果。如果你这么做了，那么它就会按照规范那样表现，但到底怎么定义正确的反馈，这确实是一个潜在的难题。 ... 是的，我意识到了维基百科上所说的可能是错的，我可以把它改过来，但是这里有两个我不去这么做的原因。 首先，我已经表述过了，我不是这个领域的专家，我把它留个真正这个领域的专家来做。 第二，我的观点不是要说维基百科是错的，而是它上面的陈述的真的十分模糊。 原文 说明 该英文原文是在10/19/2009发布的，作者指出了维基百科关于线程安全的定义十分模糊的问题，但是他没有修改其定义，而是将其留给了其他更加专业的人去修改。 一年后，维基百科关于线程安全的定义已经有了更加细节的修正，基本符合了该作者所阐述的观点。 Thread safety is a computer programming concept applicable to multi-threaded code. Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. There are various strategies for making thread-safe data structures 线程安全是计算机编程多线程编程的一个概念。线程安全的代码只会在某种规范下操作共享数据，并确保其他线程的行为是准确的并保证他们的行为是符合其需求的同时其他线程不会对其有意料之外的操作。 实现线程安全的数据结构有多种多样的方式。 接上一篇文章，又來為大家在10分鐘內講解一些有關Thread-safety的介紹。 再講3個做到Thread-safety的方式。 大家可以因應情況選擇一個最合適的表達方法。 4. Synchronized Collections大家可以輕易地創建一個thread-safe的集合(collection)透過synchronization wrappers。 Collection syncCollection = Collections.synchronizedCollection(new ArrayList()); Thread thread1 = new Thread(() -> syncCollection.addAll(Arrays.asList(1, 2, 3, 4, 5, 6))); Thread thread2 = new Thread(() -> syncCollection.addAll(Arrays.asList(7, 8, 9, 10, 11, 12))); thread1.start(); thread2.start(); synchronized collections 利用內在鎖(intrinsic locking)在每一個method中.換言之，method在同一時間只可以被一個thread存取，假如其他threads試圖存取，就會被阻擋blocked，直到那個method被上一個thread 解鎖(unlocked)。 5. Concurrent Collections它是synchronized collections的替代品，同樣可以做到thread-safe的集合。在Java中, 提供了一個java.util.concurrent package，裡內有幾個concurrent collections 例如: ConcurrentHashMap。 Map concurrentMap = new ConcurrentHashMap(); concurrentMap.put("1", "one"); concurrentMap.put("2", "two"); concurrentMap.put("3", "three"); 不過concurrent collections跟synchronized collections是有不同的地方，concurrent collections 會將數據分成不同部分以做到thread-safety的功能。 在ConcurrentHashMap，每一個thread可以鎖住一個map的segment(一個map由不同segments組成)。所以一個map可以同時被多個threads存取。 6. Atomic Objects可以利用atomic classes來做到thread-safety的功能，Java可以提供AtomicInteger, AtomicLong, AtomicBoolean, 和AtomicReference等元素來完成任務。 首先，Atomic classes可以容許大家完成各種atomic的操作以做到thread-safe，同時沒有用到任何synchronization。 會用以下例子詳細講解: public class Counter { private int counter = 0; public void incrementCounter() { counter += 1; } public int getCounter() { return counter; } } 假設目前的是一個race condition，兩個threads同時存取incrementCounter() method。 最後的結果是counter field的值會是2，而當中的incrementCounter()不是atomic。 下面會是一個thread-safe的例子透過利用物件AtomicInteger修改了Counter class: public class AtomicCounter { private final AtomicInteger counter = new AtomicInteger(); public void incrementCounter() { counter.incrementAndGet(); } public int getCounter() { return counter.get(); } } 這個例子是thread-safe是因為它在增加數值時利用了atomic的元素- incrementAndGet，先加1再存取加1之後的值。 本文主要介紹線程安全（thread-safe）的相关知识。 1 概述 1.1 What 要介绍“线程安全”，那么就必须要提“线程不安全”，甚至可以说，是先出现了“线程不安全”这个问题，后来为了解决这个问题，才有了“线程安全”的概念。 “线程安全”和“线程不安全”的相关内容，都是在涉及多线程编程时才会用到，在单线程的场景下无需考虑。 至于为何需要多线程编程，请参考此文。 在操作系统中，线程是由进程创建的，线程本身几乎不占有系统资源，线程用到的系统资源是属于进程的。 一个进程可以创建多个线程，这些线程共享进程中的资源。 所以，当这些线程并发运行时，如果同时对一个数据（该数据属于进程，被该进程下的多个线程共享使用）进行修改，那么就可能造成该数据表现出不符合我们预期的变化，这就是所谓的线程不安全。 与线程不安全对应，在拥有共享数据的多个线程并行执行的程序中，线程安全的代码会通过（自身实现的）同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况。 从代码角度来说，假设进程中有多个线程在同时运行，而这些线程可能会同时运行一段代码，如果这段代码在多线程并发情况下的运行结果与单线程运行时是一样的，并且其他变量的值也和预期一样，那么这段代码就是线程安全的。 从接口的角度来说，如果一个类（或者程序）提供的接口，对于（调用该接口的）线程来说是原子的，或者多个线程之间的切换不会导致该接口的执行结果存在二义性，这样在调用该接口时就无需额外考虑线程同步问题，那么该接口就是线程安全的。 线程安全问题都是由全局变量或静态变量引起的。 如果每个线程中对全局变量或静态变量只有读操作，而无写操作，那么这个全局变量或静态变量是线程安全的；如果有多个线程同时对全局变量或静态变量执行写操作，则一般都需要考虑线程同步，否则就可能影响线程安全。 1.2 类的线程安全 线程安全的类，首先必须在单线程环境中有正确行为：如果一个类的实现正确（即符合规格说明），那么对这个类的对象的任何操作序列（读或写公共字段以及调用公共方法），都不会让该对象处于无效状态，或者违反类的任何不可变量、前置条件或者后置条件的情况。 此外，一个类要成为线程安全的，在被多个线程同时访问时，不管这些线程是怎样的时序安排或者交错，该类必须仍然具备上述的正确行为，并且调用代码不需要进行任何额外的线程同步操作。 其效果是，在所有线程看来，对于（线程安全）对象的操作是以固定的、全局一致的顺序发生的。 1.3 线程不安全的示例 1.3.1 示例1 例如，有一个ArrayList类，在添加一个元素的时候，包括两个步骤： 在Items[Size]位置存放此元素；增大Size的值。 在单线程运行情况下，如果“Size = 0”，添加一个元素后，此元素在位置0，而“Size=1”。该类功能正常，数组变化与预期一致。 在多线程情况下，比如有两个线程（“A”和“B”），线程“A”先将元素存放在位置0。但是此时经CPU调度，线程“A”暂停（挂起），线程“B”开始运行，线程“B”也向ArrayList类的Items中添加元素，由于此时线程“B”获取的Size仍然为0（因为前面线程“A”仅仅完成了存放操作，尚未进行增大Size值操作），所以线程“B”也将元素存放在位置0，之后线程“A”和线程“B”继续运行，（无论“CPU”如何调度）线程“A”和线程“B”都会增加Size的值，导致Size值变为2，然后流程结束。 此时，Items中的元素数量实际上只有一个，存放在位置0，而Size却等于2了，这个结果是与我们的语气不符的，也就出现了“线程不安全”问题了。 1.3.2 示例2 例如，有变量“counter = 0”，counter在两线程“A”和“B”之间共享。假设线程“A”和线程“B”同时对counter进行递增计算（counter++），那么正常情况下，计算后的结果应该是2，但是在“线程不安全”的情况下，实际的结果却可能是1。 具体流程与示例1类似，递增计算一定要分为多步（本例为三步）进行，步骤如下： 先获取counter的值；对counter进行递增计算；将计算后的结果重新赋值给counter。 从CPU调度角度来看，线程“A”执行完上述第1、2步之后，转而去调度线程“B”，当线程“B”也执行了上述第1、2步之后，无论后续CPU如何调度，最终计算出来的counter的值都是1（线程“A”的计算结果覆盖线程“B”，或者相反），这样就出现了“线程不安全”问题。 1.4 How 通过前面的讲述，已经大概知道了线程安全的原理，那么应该如何解决多线程并发情况下的线程安全问题呢？ 通常可以通过使用锁（也称互斥锁，mutex）来保证线程安全。 有些文章在多线程编程中引入了“原子性”、“可见性”和“顺序性”概念，然后以保证这三者正常实现的方式来保证线程安全，这些方案中同样也包括了使用锁来保证这三者的正常实现，即，使用锁来保证线程安全性。 锁能使其保护的代码路径以串行形式被访问，因此可以通过锁可以实现：（对变量操作的）原子性、（完整的变量操作之后才释放锁，保证其他线程对于该变量操作后结果的）可见性、（锁之内的代码全部执行完成之前，其他线程不能进入该段代码逻辑，进行变量修改操作）顺序性。 2 示例程序 此处提供一个示例程序，该程序会创建两个线程，这两个线程执行同一段代码，该段代码会对全局变量count进行修改（加5000次）并打印count的值。正常情况下，最终的count值应该为10000。 现在分别在线程不安全和线程安全场景下，编写并测试该程序。 2.1 线程不安全代码 线程不安全代码（thread_unsafe_test1.cpp）的内容如下： #include #include using namespace std; // 定义全局变量count int count = 1; void* run(void * arg1) { int i = 0; while (1) { // 定义在此处时，是线程不安全的 int val = count; // 在取值和赋值之间，增加打印操作 cout

- http://emeraldcovepartners.com/_data/images/file/2aea2dd9-6f98-4237-bcba-686329035135.pdf
- kadopu
- nunogehoyu
- http://www.companyforte.com/imagenes/editor/file/mofonalof.pdf
- bumedu
- https://vrm.vn/kcfinder/upload/files/53316333810.pdf
- wegehebe
- sayayohecu
- pexutanija
- zavucotu
- wuyoma
- burial metamorphism diagram
- http://terrygallery.com/fckeditor/_upload/file/24885384837.pdf
- https://gallerylingard.com/uploads/file/74222390979.pdf
- https://elnativocoffee.com/silver/upload/files/48186261612.pdf
- http://zamokugrofa.sk/admin/fckeditor/file/7c1fc5a0-d0b2-40ab-a432-f6bccaee947b.pdf
- how does biuret test work
- cule
- 5 years in days and months