Fault Diagnosis Method and Software Development Research

Liya Cai

Suzhou Industrial Park Institute of Service Outsourcing, Suzhou, Jiangsu, 215000, China

Keywords: Fault Diagnosis, Software Development, Software System

Abstract: Software fault diagnosis is an important part of software development and maintenance. With the wide application of computer software technology, the requirements for software system performance and reliability are getting higher and higher. In response to this problem, based on work experience, the commonly used software fault diagnosis methods are discussed.

1. Introduction

Software troubleshooting is an important part of software development and maintenance. With the wide application of computer software technology, the requirements for software system performance and reliability are getting higher and higher, especially in medical, aerospace and other aspects, any tiny software failure can bring extremely serious consequences. . Therefore, software quality assurance has become increasingly important. To ensure the quality of software, we must start from the source, that is, in the stage of software design and development, we must carefully study and carefully code, and strive to be strict and readable, so as to minimize software failure. It also reduces the maintenance cost of the system in the future. After the software development is completed, it needs to undergo comprehensive testing, including unit testing, integration testing, etc., to discover problems left in the design or coding. Unfortunately, no matter how hard we try, software failures seem to never be avoided. All we can do is to avoid major failures and minimize small failures. An important way to achieve this goal is to perform software troubleshooting for problems found in the test or problems found in actual operation after the software development is completed. It is an extremely important skill in software development and maintenance, and is a skill that many software developers, especially those who have just entered the job, need to learn. Although troubleshooting is so important, we rarely see discussions about these issues. Due to the differences in software systems themselves, different systems have different software diagnostic methods. Therefore, many developers often take a long time to find a feasible fault diagnosis solution.

2. Single problem and its diagnosis

Software failures are often encountered in software development and maintenance. According to fault types, they can be roughly divided into null pointer access, array out of bounds, stack overflow, infinite loop, file problem, lock protection problem, message loss, queue overflow, memory leak, memory. Repeat release, memory rewrite, etc. In addition, whether or not the problem can be reproduced can be divided into an easy reproduction problem, an irregular reproduction problem, and an extremely difficult reproduction problem. In general, different types of software failures are slightly different depending on whether the problem can be reproduced or not. Diagnosing null pointer access is generally simple. First, analyze whether the null pointer is normal. For normal null pointers, just increase the protection, that is, confirm that the pointer is not empty. According to our experience, if such problems occur, they will generally appear in many places. Therefore, it is necessary to "except for the evils" to ensure that the same type of problems in the file or function are completely modified. If the null pointer here is illegal, you should check if there is a problem with the caller of the function. For example, the caller forgets to assign a value to the pointer before calling the function, and the function call sequence has problems, etc., and it is often necessary to view it step by step. The diagnostic method of array out of bounds is similar to null pointer access, focusing on the detection of array subscripts. If we can't make enough restrictions on the caller, we need to increase the limit on the array index in the called function. In addition, an important manifestation of an array out of bounds is memory rewriting, that is, the out-of-bounds array of the array causes the memory behind the array to be overwritten, which we will discuss in the Memory Overwriting section.

Stack overflows are generally caused by recursive calls. Since each function is called once, we need to allocate space in the memory stack area for storing function variables, return values, etc., so if there are too many recursions, it will cause a stack overflow. For such problems, it is generally necessary to detect the calling relationship between functions, and to detect what causes the function to recursively call abnormal exit. Such problems are generally caused by improper handling of the hook. The infinite loop is generally caused by inadvertent handling of the boundary condition. For example, the loop exit condition is to test that an unsigned number is not less than zero. In fact, this is an eternal true. It is also very important that some systems check the continuous running time of each task in order to prevent a task from monopolizing C P U for a long time, and restart the system through the watchdog when it reaches a certain time. This type of problem manifests itself as an infinite loop, which may actually be due to too many cycles in the task, or a single loop execution time that is too long, causing the watchdog to be reset before the system loop is executed. Such problems can generally be added to the run-time check in the loop, and the CPU resources are actively released when the task runs for a certain period of time. In multitasking systems, in order to protect critical resources of the system, we generally need to introduce a lock mechanism, but the use of locks is also an important cause of system problems. The lock problems we often encounter in our work generally include deadlock problems and critical resources are not effectively protected. For the deadlock problem, all the resources in the system should be divided into several levels. Firstly, the high-level lock is allocated and then the underlying lock is obtained. Try to avoid the low-level lock and then obtain the reverse operation of the high-level lock. For the problem that critical resources are not effectively protected, it is necessary to detect the use of locks and ensure that access to all critical resources is placed within the protection scope of the lock.

Memory re-release means that the system tries to release the memory that has been released. This is a very serious problem, and it has been found that such problems indicate that the system is using a pointer that has already been released. At this point, if you don't release the memory, but modify the contents of the memory, it will lead to a very difficult problem to diagnose - memory rewriting, we will discuss this in the next section. For repeated release, it is relatively easy to solve some problems. We can get a lot of information from the repeated release prompts, such as the size of the memory released repeatedly, the position of repeated release, verify whether the release is reasonable, whether there is any other place to release the memory, and verify The relationship between them, in turn, identifies the source of the problem that causes the memory to be repeatedly released here. In general, in order to solve such problems more easily, many systems will add the call stack information when the memory is released, so that when the memory is repeatedly released, it can locate where the memory was last released. Conducive to the solution of the problem. Memory rewriting is a more difficult type of problem in troubleshooting. According to the reason of memory rewriting, it can be divided into memory over-the-air access class memory rewriting, memory repetitive release class memory rewriting, wild pointer class memory rewriting and so on. From the difficulty level, the first two types of memory rewriting problems are easier to solve, and the wild pointer class memory rewriting problem is more difficult to solve. In theory, if you can first determine the type of memory rewriting, it will be very beneficial to solve such problems. Unfortunately, it is generally only when you find the source of the problem to determine which type of memory rewriting. Habitually, we often make an assumption about such problems in advance, that is, according to the difficulty of the problem, first assume that it is a memory over-the-counter access class memory rewriting. So we need to analyze the memory information before the memory is rewritten, that is, to see where the previous memory is applied, the size of the memory, whether there is any possibility of cross-border access. If we exclude the situation where the previous memory is out of bounds, we can further suspect whether it is caused by repeated memory release.

At this point, you can check if the memory of this block has been released. If so, this memory may be applied for and modified by other modules, so memory rewriting will occur when accessing.

3. Comprehensive problems and their diagnosis

The classification and diagnosis methods of single problems are analyzed above, but the problems encountered in reality are often not single problems, but comprehensive problems. The comprehensive problem involves a wider and more complex, seemingly incompetent, but there are rules to follow. The following is a more suiTable method for analyzing comprehensive problems summarized by the author.

It is very difficult for us to find the problem in the first place when the problem occurs. For example, memory rewriting, if a block of memory is rewritten at a certain time, the problem can only be found when accessing the rewritten memory, and the distance from the crime scene may have been a long time. When analyzing these problems, you need to find the symptoms of the problem - the initial anomaly, which is often the key to whether the problem can be solved as soon as possible or even finally resolved. Therefore, for the comprehensive problems encountered, after clarifying what problems are currently occurring, it is necessary to analyze the system logs, debug information, etc., and collect as many prior exception information as possible.

After collecting enough information, start analyzing the problem. There are generally a number of options that can lead to the final problem. The first step is to boldly guess the suspicious scheme and enumerate all the suspicious circumstances. Even if the probability of a certain scheme is small, in extreme cases, they may be the source of the problem.

After listing all suspicious scenarios, you need to filter the case. You can generally sort by the probability that a suspicious scenario can occur and analyze the most suspicious scenarios. When sorting the case probability, pay attention to the analysis of the collected abnormal information. In many cases, the problem does not happen suddenly, but gradually. Combining early anomaly analysis with screening suspects can greatly speed up the analysis of problems. It can lead to early anomalies or suspicious schemes triggered by some prior anomalies that are likely to be the source of the problem, so these suspicious schemes can be given higher priority in the analysis.

After sorting suspicious cases, you need to analyze the cases one by one. At this point, it is still necessary to analyze the abnormal situation before the problem. Imagine the steps through which these anomalies ultimately lead to problems. In some cases, it may not be enough to rely on these exceptions to restore the crime scene. At this point, you can speculate on the conditions that need to be solved, and then go through the code, or make assumptions to find these missing conditions, and try to prove the feasibility of these conditions. Sex, which ultimately restores the scene and finds the source of the problem. It should be noted that the order of the above processes is not fixed. After some problems arise, the crime scene may be destroyed immediately. It is generally very important to follow the above steps. Otherwise, the first scene may be lost quickly without collecting enough data that is very important for analyzing the problem. For some simple situations or problems that are easier to reproduce, you can first analyze the suspicious schemes, roughly arrange these schemes and start the analysis from the most suspicious schemes, and then collect information according to these schemes.

In fact, comprehensive problems are often very complicated. Many problems are often hidden. The above methods cannot tell you the answers to the questions, but they give practical analysis steps for most of the problems you can encounter, but different. The difficulties and priorities of the above steps may be slightly different.

4. Conclusion

This article discusses the basic problems of common software and the diagnostic methods for comprehensive problems. And combined with examples, explain how to use our method to analyze more difficult comprehensive problems. I hope that readers, especially software practitioners who have just taken up their jobs, can learn from them.

References

[1] Li Shengshan. Design of Hydraulic Transmission Control Based on PLC and FluidSIM Software [J]. Mechanical Management and Development, 2014, (4): 42-44.

[2] Jia Guangzheng, Wang Jindong, Yang Songshan, et al. Discussion on the experimental teaching mode of "hydraulic and pneumatic"[J]. Machine Tool & Hydraulics, 2007, (2): 151-152.

[3] Meng Qingyun. Reform and Practice of Hydraulic and Pneumatic Experiment Teaching [J]. Hydraulic & Pneumatics, 2011, (3):17-19.

[4] Bi Changfei. Application of FluidSIM 3.6 Simulation Software in Hydraulic and Pneumatic Technology [J]. Hydraulic & Pneumatics, 2011, (8):111-114.

[5] GUO Lianjin, PAN Bin. Application of FluidSIM in Simulation Experiment of Hydraulic and Pneumatic Control [J]. Experimental Technology and Management, 2015, (8):121-126.