Makacs hibák megkeresése sokatmondó hibainformációkkal

Amikor egy felhasználó egy olyan hibát jelez, amit nem tudunk megismételni, bízzuk inkább a programunkra a hiba megkeresését. Hozzunk létre egy eseménynaplót, és legyünk hibakereső bajnokok a szoftver alkalmazása után.

Egy hiba nyomon követése gyakran a programfejlesztés egyik legbonyolultabb folyamata. A felhasználók sokszor a fejlesztőétől eltérő helyzetben használják a programot, és a programhibák, amik a felhasználónak nagy gondot okoznak, a fejlesztő gépén esetleg meg sem jelennek. Néha a hibák maguktól elmúlnak, vagy a hálózatos programokban csak akkor jelentkezik a tünet, amikor egy bizonyos kiszolgálóval vagy ügyféllel folytatunk adatcserét. Ebben a cikkben olyan módszereket ismertetek, amelyek segítségével a programfejlesztők könnyebben kinyomozhatják egy hibajelenség eredetét.
Először bemutatok két módszert a hibajelentések könnyebb fogadására és kezelésére, majd megmutatom, miképpen érhetjük el, hogy a programunk jobban használható hibakereső kimenetet adjon. Ezután szólni fogok a kellemetlen hibák felderítéséről, végül bemutatok néhány fogást arra, hogyan előzhetjük meg a programhibák kialakulását.
A cikkben bemutatandó eljárások közül sokat alkalmaztam az ***OfflineIMAP*** fejlesztésekor.

A programhibák felderítése

Mielőtt megvizsgálnánk, hogyan tehetünk szert jobb hibajelentésekre, nagyon fontos, hogy biztosak legyünk benne, hogy egyáltalán foglalkozni tudunk-e a kapott hibajelentésekkel. Kisebb projektek esetében elegendő lehet, ha egyszerűen egy e-mail címet tartunk fenn erre a célra, habár a legtöbb projekt esetében ennél többre van szükség. A fejlesztők gyakran elfoglaltak és feledékenyek. A hibák kijavítása meglehetősen bonyolult is lehet, és több ember közreműködését is igényelheti, de az is előfordulhat, hogy elárasztanak a hibajelentések.
A ***hibakövető rendszerek (BTS, bug-tracking system)*** remek eszközt biztosítanak ahhoz, hogy egy hibáról se feledkezzünk el. A legtöbb BTS lehetőséget ad a levelezés követésére, a csatolt állományok kezelésére és a hibákkal kapcsolatos feladatok megosztására. Néhány támogatja a hiba súlyosságа, a felhasználói környezet és bizonyos összetevők alapján történő osztályozást is.
Amennyiben a projektünket a ***SourceForge*** vagy ***Savannah*** weboldalakhoz hasonló tárhelyeken helyezzük el, akkor ezzel biztosítva van számunkra egy ***BTS***-rendszer is. Ezt mindenképpen érdemes használnunk és a felhasználóinkat is arra kell ösztönöznünk, hogy a levelezőlisták helyett ezt a felületet használják a hibák jelzésére.
Ha ennél nagyobb rugalmasságra tartunk igényt, kereshetünk a ***Linuxhoz*** készített ***BTS***-rendszert is. Néhány a legnépszerűbb ingyenes ***BTS***-programok közül:

- ***Bugzilla***: A ***Mozilla Projekt*** által használt ***BTS*** egy rugalmas rendszer, amelyet elsődlegesen a webes felületén keresztül alkalmazhatunk.
- A ***Request Tracker*** használható akár hibakövető, akár támogatás-nyilvántartó rendszerként. Rendelkezik webes és levelezői felülettel is, bár bizonyos rendszergazdai funkciók csak a webes felületen keresztül érhetőek el.
- A ***Jitterbug*** a ***Samba Projekt*** által is használt ***BTS***. Az alkalmazott elv hasonló a Bugzillaban használthoz, de annál kisebb programról van szó.
- A ***Debbugs*** a ***Debian Projekt*** által használt ***BTS***. A ***Debbugs*** rendelkezik egy webes felülettel, de ez csak olvasható, minden művelet e-mail-en keresztül történik. A ***Debbugs*** a nagy projektekhez a legmegfelelőbb az egyértelműen azonosítható összetevőivel és ezek hatáskörével.

Tegyük egyszerűbbé a hibák bejelentését

Előfordul, hogy kellemetlen hibát találok egy programban és erről tájékoztatni akarom a fejlesztőt, de ehhez egy részletes kérdőívet kell kitöltenem és olyan dolgokat megadnom, amiket nem nagyon szeretnék. Egyszerűvé kell tennünk a felhasználók számára a hibák és a felderítésükhöz szükséges információk elküldését. Ha a bejelentéseket a Világhálón keresztül fogadjuk, a folyamat legyen minél egyszerűbb. Ne követeljünk túl sok információt és olyan bejelentéseket is fogadjunk el, amikről esetleg hiányzik néhány adat. Ne várjuk a felhasználóktól, hogy tisztában legyenek a program különböző összetevőivel, vagy hogy tudják, melyik fejlesztőnek kell foglalkoznia az adott problémával.

A napló használata

Problémák felderítése közben gyakran szeretnénk tudni, hogy milyen környezetben működik a program, máskor

pedig esetleg arra van szükség, hogy ismerjük a hiba jelentkezését megelőző, azt kiváltó eseményeket. Mivel a felhasználók nem feltétlenül ismerik az általunk kódolásra és hibakeresésre használt eszközöket, elterjedt a naplózás használata erre a célra. A naplózás egyszerűen egy rekord rögzítését jelenti az elvégzett műveletekről. Az egyszerű programok esetleg csak a képernyőre írják az adatokat, de valószínű, hogy ennél kicsit jobb eszközre lesz szükségünk.

A nem interaktív programok – amilyenek például a hálózati kiszolgálók – nem rendelkeznek olyan kijelzővel, amin megjeleníthetnék ezeket az információkat. Ezek a programok a bejegyzéseiket rendszerint naplófájlba rögzítik, vagy a ***Linux*** és ***UNIX*** rendszerekbe beépített ***syslog*** eszközt használják.

Az interaktív programok vagy a képernyőre vagy egy fájlba írhatják az információkat. A naplófájl megkönnyíti a hibajelentések elkészítését, mert a felhasználó egyszerűen csatolhatja ezt a fájlt a bejelentéséhez.

Sokszor meglehetősen sok adatra van szükség annak megállapításához, hogy pontosan mi is történik egy adott rendszeren, de ez az adatmennyiség lehetetlenné is teheti a normál működést, elboríthatja a felhasználó képernyőjét, vagy megtöltheti a merevlemezét. Éppen ezért sok programban találkozhatunk a naplózási szint fogalmával, ami annyit jelent, hogy a felhasználó futási időben meghatározhatja a naplózott információ mennyiségét. Néhány program osztályozza is az eseményeket, így a felhasználó azt is megadhatja, hogy milyen típusú információk kerüljenek be a naplóba. Az ***OfflineIMAP*** ezt a megközelítési módot használja. Kellemetlenebb problémák esetén a felhasználó bekapcsolhatja a kapcsolattartási naplót, amely minden, az ***IMAP***-kiszolgáló felé elküldött vagy onnan érkező adatot naplóz.

A ***Python 2.3*** változatában megjelent egy hasznos modul, aminek ***naplózás (logging)*** a neve. Ez a modul egységes felületet biztosít az üzenetek naplózásának számos különböző módszeréhez. A támogatott naplózási módszerek közt megtaláljuk a naplófájlok használatát, a hálózati szolgáltatásokat, a ***syslog***-ot, az e-mail üzeneteket és számos egyéb módszert. Nézzünk egy egyszerű példát a naplózó modul használatának bemutatására:

```
#!/usr/bin/env python
import logging, sys
# A naplózó objektum létrehozása
l = logging.getLogger('testlog')
# Egy kezelő létrehozása és az objektumhoz rendelése
handler = logging.StreamHandler(sys.stderr)
l.addHandler(handler)
# A szintek:  DEBUG, INFO, WARNING, ERROR,
# CRITICAL.
# Beállítjuk az alapértelmezett szintet.
# Az e szint alatti
# naplóüzeneteket figyelmen kívül hagyjuk.
l.setLevel(logging.INFO)
# Kipróbáljuk.
l.debug("Debug message -- system initialized.")
l.info("Here's some info.  I've just debugged.")
l.warning("I don't have many messages left.")
l.error("Only one more message to go.")
l.critical("Nothing else to do!")
```

A program a naplózás alapbeállításával kezdődik. A naplóüzenetek szabványos hibakimenetre történő kiírását a `StreamHandler` kezelő végzi. A naplózás szintjét `INFO`-ra állítjuk. Öt naplóüzenetet naplózunk, de a program futásakor csak az utolsó négyet látjuk. A `debug` üzenetet a naplózási szint `INFO`-ra állítása kiszűrte. Sok program biztosít beállítási lehetőséget vagy parancssori paramétert a naplózási szint futásidőben történő beállítására. Ettől eltérő naplózási módszert is használhatunk egyszerűen egy másik kezelőt adva a `Logger` objektumunkhoz. A ***Python*** leírásában megtaláljuk az összes használható kezelő ismertetését.

A bemenet ellenőrzése

Bizonyosodjunk meg arról, hogy a kapott bemenet megfelelő. Például ha a parancssoron keresztül várunk valamit, ellenőrizzük a paraméterek megfelelő számát, mielőtt megpróbálnánk alkalmazni (vagy a kapott kivételt elcsípni). Ezzel a módszerrel jobb hibaüzenethez juthatunk. Íme egy ***Python*** példaprogram ennek bemutatására:

```
#!/usr/bin/env python
import sys
try:
    print "You supplied: %s" % sys.argv[1]
except IndexError:
    print "You forgot an argument."
```

A kivételek kezelése

Számos programozási nyelv – többek közt a ***Java***, ***Python*** és az ***OCaml*** – biztosít lehetőséget a kivételek kezelésére. A kivételek használatával a kívánt helyen csíphetjük el a hibákat ahelyett, hogy minden esetlegesen problémát okozó hívást ellenőriznünk és az előforduló hibákat kezelnünk kellene. Időnként nem okoz gondot a kivételek lekezelés nélküli átengedése, de rendszerint nem ez a helyzet, a kivételeket el kell fogni és megfelelően kezelni. Habár megfelelő megoldás lehet felfüggeszteni a program működését, amikor nem tudjuk megnyitni a felhasználó által kért fájlt, de ezt jobb egy megfelelő hibaüzenettel tennünk, amely megadja a probléma jellegét és a fájl nevét, ahelyett, hogy a felhasználó csak a kivétel csúnya üzenetét kapná meg.

A kivételek elfogása

Még a tényleg végzetes kivételek esetén is érdemes lehet a kivételt elfogni. Ez lehetővé teszi például a kivétel naplófájlba való írását vagy egy felugró ablakban a grafikus felületen történő megjelenítését. Ez megkönnyíti a felhasználók számára, hogy a rögzített nyomokat visszaküldjék számunkra. Használhatunk általános kivételelfogót is, amely egyéb műveleteket is végrehajthat, például az átmeneti tárak rögzítését, amely megkönnyíti az események visszakövetését. Az alábbi példa minden kivételt elkap és naplóz néhány egyéb, a futó programmal kapcsolatos információval együtt. Ezután újra kiváltja a kivételt és kilép.

```
#!/usr/bin/env python
import logging, sys, StringIO, traceback, os
l = logging.getLogger('testlog')
handler = logging.StreamHandler(sys.stderr)
l.addHandler(handler)
```

```
formatter = logging.Formatter("LOG: %(message)s")
handler.setFormatter(formatter)
l.setLevel(logging.INFO)
def logexception():
    sbuf = StringIO.StringIO()
    traceback.print_exc(file = sbuf)
    excval = sbuf.getvalue()
    l.critical(" *** Exception Detected ***")
    l.critical("Current PID: %d" % os.getpid())
    l.critical("Program name: %s" % sys.argv[0])
    l.critical("Command line: %s" % \
                str(sys.argv[1:]))
    for line in excval.split("\n"):
        l.critical(line)
def main():
    print "Hello, I'm running."
    raise RuntimeError("Oops! I've had a problem!")
try:
    main()
except:
    logexception()
    raise
```

A program futtatásakor valami ilyesmit kell látnunk a képernyőn:

```
Hello, I'm running.
LOG:  *** Exception Detected ***
LOG: Current PID: 28441
LOG: Program name: /tmp/logerror.py
LOG: Command line: []
LOG: Traceback (most recent call last):
LOG:   File "/tmp/logerror.py", line 30, in ?
LOG:     main()
LOG:   File "/tmp/logerror.py", line 27, in main
LOG:     raise RuntimeError("Oops! I've had
         ➥a problem!")
LOG: RuntimeError: Oops! I've had a problem!
LOG:
```

A kivételkezelő megtalálta a kivételt, begyűjtötte a kapcsolódó információkat és sikeresen rögzítette a naplóban. Másodszor is láthatjuk a ***visszakövető (traceback) információkat***. A program végén lévő `raise` utasítás újra előidézi a kivételt és lehetővé teszi annak normál módon történő lekezelését. Ez azt jelenti, hogy egy visszakövetéssel félbeszakítja a programunkat. Az elvárásainktól függően egy `sys.exit()` műveletet is használhatunk ehelyett.

A bejelentett hibák megkeresése

Most, hogy már rendelkezünk néhány módszerrel arra, hogy a felhasználókat segítsük a minél jobb hibajelentések elküldésében, vizsgáljuk meg ezeknek a jelentéseknek a felhasználási módjait. Egy hibanaplóval és talán a visszakövetési információkkal felfegyverkezve a következő kérdéseket tehetjük fel magunknak:

- Elő tudom-e idézni a hibát a saját futtatókörnyezetemben? Ha ez a saját gépünkön is sikerül, közel kerültünk ahhoz, hogy ki is javítsuk. Használjunk egy hibakeresőt vagy valamilyen más eszközt az ok megkereséséhez.
- Az elvártnak megfelelő volt a bemenet és a kimenet? Előfordulhat, hogy a felhasználó olyan értéket használt, amire nem gondoltunk a program írásakor. Esetleg egy hálózati ügyfélgép vagy kiszolgáló kezel egy kicsit eltérően valamilyen protokollt. Talán csak a bemenet vagy a kimenet maga torzult és a hiba nem is a mi programunkban van. Nagyon hasznos tud lenni ezen a ponton egy olyan hibakereső napló, amely az összes be- és kimenetet tartalmazza.
- A vártnak megfelelően alakult a program futási folyamata? Amennyiben a naplónk különböző függvények és eljárások hívását tartalmazza, szükséges, hogy egy programmal követni tudjuk a futási folyamatot. Előfordulhat, hogy valamilyen feltételek együttesen vezettek egy élő kódrészlet figyelmen kívül hagyásához, amely később vezetett a hibához.
- Mi a helyes futás utolsó pontja? Ez lehet a hibát közvetlenül megelőző rész, de az is előfordulhat, hogy már a hibát valamennyi idővel megelőzően történt egy rossz adatátadás. Ha megkeressük azt a legkésőbbi helyet, ahol a program még megfelelően működött, akkor könnyebben behatárolhatjuk a félresiklás pontos helyét.
- Ha kéznél vannak a visszakövetési információk (traceback), ellenőrizzük, hogy a verem tartalma megfelelőnek tűnik-e. Vizsgáljuk meg, hogy a függvényhívások és a kapott paraméterek a vártnak megfelelőek-e.

A hibák megelőzése

Ugyan hasznosak a fent ismertetett módszerek, de önmagukban még nem elegendőek. Fontos az is, hogy olyan fejlesztési gyakorlatot folytassunk, amely segít csökkenteni a hibák előfordulási valószínűségét. Néhány megfontolásra érdemes módszer ezek közül:

- Alkalmazzunk az ***egységek tesztelését (unit testing)***. A ***Java***, ***Python***, ***OCaml***, ***Perl*** és ***C*** nyelvek mindegyike rendelkezik egységtesztelő keretrendszerrel. Használjuk ki ezeket és a lehető legtöbb futási esetet vizsgáljunk meg velük. Ez különösen az olyan nyelveknél fontos, mint a ***Python***, amelynél egy bizonyos futáskor még a kódértelmezés sem terjed ki a teljes kódra. Fontos ez a Java esetében is, ahol a futásidejű kivételek nem megfelelő objektumváltást eredményezhetnek.
- Kerüljük a globális változókat. A globális (vagy osztályszinten globális) változók segítenek a problémák körülhatárolásában és megóvnak a többszálú programok szinkronizációs gondjaitól. A globális változók nem várt és nehezen visszakövethető mellékhatásokat okozhatnak a függvényhívásokban.
- Használjuk a legmegfelelőbb eszközt az adott munkához. A programozási nyelvek mindegyikének megvan a maga erőssége és gyengéje, nincs olyan nyelv, ami minden feladatra a legmegfelelőbb. Például ***Perlben*** programozva könnyű a körülhatárolt szövegfájlok szabályos kifejezésekkel történő elemzése, az ***OCaml*** pedig olyan eszközökkel rendelkezik, amelyeket kifejezetten ***fordítóprogramok (compiler)*** írására használhatunk. Az egyik nyelvben könnyen kifejezhető probléma sokkal bonyolultabb lehet egy másikban.
- Ne használjunk túl sok különböző eszközt sem. A legtöbb projektnek előnyére válik a használt eszközkészlet

behatárolása. Válasszuk ki a legmegfelelőbb nyelvet és programkönyvtárat és csak akkor nyúljunk új eszközhöz, ha erre alapos okunk van.

- Használjunk karakter- és memóriakezelő programokat. Sok nyelv, többek közt a ***Java***, ***Python***, ***OCaml***, ***Perl*** és ***Ruby*** háttérben futó memóriakezelést biztosít, így nincs szükség a memória lefoglalására és felszabadítására. Nem kell az explicit karakterlánc-végződésekkel és a karakterláncokra vonatkozó hosszkorlátokkal foglalkoznunk, melyek mindegyik gyakori probléma a *C* nyelv használatakor és futásidejű hibákhoz, vagy biztonsági résekhez vezethet. Ha mindenképpen *C*-t kell használnunk, fontoljuk meg egy hulladékgyűjtő- vagy memóriatartalék-programkönyvtár használatát.
- Először tegyük a programot működővé és utána keressük meg a lehető legjobb megoldást. Sok esetben jobb kifejleszteni egy működő kódot és később optimalizálni. Sokan először optimalizálnak, ami működik is olykor, mégis rendszerint fontosabb egy egyszerű, hibátlan kód, mint egy olyan, ami a lehető leggyorsabb.
- Kódoljunk tisztán, készítsünk függvényeket az egyes kódrészletekhez. Használjunk megjegyzéseket. Készítsünk leírást arról, hogy melyik függvénynek mi a szerepe és milyen hatással van a környezetre.

Esettanulmány: egy hiba az OfflineIMAP-ben

Az ***OfflineIMAP*** egy olyan program, amely kapcsolatot tart az ***IMAP***-kiszolgálókkal és összehangolja az ***IMAP*** mappaszerkezetet a helyi faszerkezettel. Sokféle ***IMAP***-kiszolgáló létezik, amelyek nem teljesen azonos módon működnek. Kétéves pályafutása során az ***OfflineIMAP*** egyre nagyobb arányban alkalmazta a cikkben bemutatott hibakereső eljárásokat. Azok a problémák, amelyekkel a felhasználók szembesülnek, gyakran előidézhetetlennek bizonyulnak az én egyedi rendszeremen, ezért elengedhetetlen, hogy a program részletes naplózást folytasson. Néha maguk az ***IMAP***-kiszolgálók sem hibátlanok, ezért a hibajelentések jelentős részénél az első megválaszolandó kérdés az, hogy egyáltalán az ***OfflineIMAP*** rendszerben lévő hibáról van-e szó. Az esetek meglepően nagy százalékában nemleges a válasz. Az ***OfflineIMAP*** számos olyan ***IMAP***-szolgáltatást használ, amelyet az ***IMAP***-ügyfelek többsége nem, és ezek a tulajdonságok néhány kiszolgálón elég gyengén tesztelt állapotban vannak.

Szeretnék bemutatni az ***OfflineIMAP*** egy különösen makacs hibáját, amivel egyszer dolgom akadt. Körülbelül egy évvel ezelőtt egy felhasználó egy hibát jelzett a programban, amelyet a ***Debian*** hibakereső rendszerével fedezett fel. Sajnos nem tudtam újra előidézni a hibát és az eredeti bejelentőnek éppen nem volt a naplózás bekapcsolva, amikor a hiba jelentkezett. Hibakereső információkat szintén nem tudott biztosítani. A kapott információk birtokában, amelyek közt egy hibaüzenet is volt, a cikkben korábban vázolt lépéseket követve sikerült némi információt összegyűjtenem. A be- és kimenet nem állt rendelkezésemre, de a program folyamata és a verem tartalma jónak tűnt. Végül már tudtam, hogy a program melyik részén következett be az esemény, de azt nem, hogy miért, így a kódban egy ideig bennmaradt a hiba. A helyzetet bonyolította, hogy a jelenség nem jött elő állandóan, a program néha jól működött, időnként azonban bedobta a törölközőt.

Később egy második felhasználó is tapasztalta ugyanezt a hibát, észrevette a korábbi debianos hibajelentést és elküldte a saját információit vele kapcsolatban. Az ***OfflineIMAP*** végzetes hiba esetén megpróbálja kiíratni a hibakereső napló részeit, ennek a felhasználónak pedig sikerült elcsípnie ezeket a kimeneteket. Az ***OfflineIMAP*** e tulajdonsága hasznosnak bizonyult tehát, mivel gyakran lehetetlen egy problémához vezető helyzet utólagos reprodukálása.

Ebben az esetben segített a kapott információ, most már képes voltam felidézni, mit csinált az ***OfflineIMAP*** közvetlenül a hiba felbukkanása előtt. Ahhoz azonban kevés volt, hogy a probléma okára is rájöjjek, minden normálisnak tűnt. A hiba csak időnként bukkant fel, és további információk nem álltak rendelkezésre.

Végül egy harmadik felhasználó is tapasztalta ugyanezt a hibát. Neki is voltak információi, de ahhoz kevés, hogy választ kapjak a kérdésemre. Valaminek még történnie kellett, ezért a kérdéses kódrészhez még részletesebb naplózást készítettem. Remélhetőleg a következő alkalommal a részletesebb információk alapján majd visszakövethetem a probléma okát.

A folyamat során számos tényező játszott fontos szerepet. Az első, hogy az ***OfflineIMAP*** végzetes hiba esetén mindig létrehoz egy használható verem-képet. Még e legkevésbé részletes jelentés is pontosan megmutatta, hogy milyen állapotban volt a program az összeomláskor. Másodszor, a hibanaplók ugyan hasznosak, de kevéssé vehetjük hasznukat akkor, amikor egy bizonyos hibát nem tudunk könnyen előidézni. A hibakereső információk kiíratása a program összeomlásakor vagy hibás működésekor hasznos lehet a probléma leküzdésében.

A hibakövető rendszer is fontos szerepet játszhat a probléma felderítésében. Mivel a ***Debian*** hibajelentései nyilvánosak, a három említett hibabejelentő felismerhette a fennálló hibajelentéseket és hozzátehette a saját információját. Ez mindenkinek segített az adott témával kapcsolatos információk kezelésében és kiindulópontot biztosított azoknak a felhasználónak, akik először botlottak a problémába.

Összegzés

Számos módja van annak, hogy segítsük a felhasználóinkat a hibák bejelentésében és visszakövetésében, de ezek önmagukban még nem elegendőek. Ne feledkezzünk meg arról, hogy minél könnyebb legyen a hiba bejelentése és követése, és hogy törekedjünk a kód tisztaságára. Végül azt se feledjük, hogy önmagában egyik lépés sem varázstöltény. Együtt alkalmazva egyszerűsíthetik a hibakereső folyamatot és segíthetnek a problémák felderítésében, de nem feltétlenül oldanak meg minden kérdést.

Linux Journal 2005. január, 129. szám

A cikkhez kapcsolódó források a ➲ ***www.linuxjournal.com/aricle/7747*** címen érhetőek el.

John Goerzen hosszú ideje Linux-programozó, a Foundations of Python Network Programming (A Python hálózati programozásának alapjai) szerzője. A szoftverágazat igazgatójaként dolgozik a Public Interest, Inc. cégnél. A hozzászólásokat a jgoerzen@complete.org címre várja.