GCOV

Bevezetés

A kódbefedés (code coverage) metrikája kulcsfontosságú a szoftverfejlesztés során, mivel segít abban, hogy jobb minőségű szoftvereket hozzunk létre, és növeli a fejlesztési folyamat hatékonyságát több módon is:

Bizalom növelése: Magas kódbefedési arányú kód esetén a fejlesztők és a projekt érdekelt felei nagyobb bizalommal tekinthetnek a kód minőségére és megbízhatóságára. Ez növeli a csapat és az ügyfelek elégedettségét, és segít megőrizni a jó hírnevet a szoftvertermékről.

Tesztkészítési hatékonyság: A kódbefedés segíthet azonosítani azokat a kódterületeket, amelyek még nem fedettek le tesztekkel. Ez segíthet a fejlesztőknek abban, hogy hatékonyabban tervezzenek és hajtsanak végre teszteket, mivel a hiányzó területekre fókuszálhatnak, és biztosíthatják, hogy a tesztek teljes körűen lefedjék a kódot.

Fejlesztési folyamat hatékonysága: A magas kódbefedési arányú kód segíthet csökkenteni a kódhibák számát és azok okozta visszahívásokat, javításokat, ami időt és erőforrásokat takaríthat meg a hosszú távon. Ezenkívül a megbízhatóbb kód kevesebb időt és energiát igényel a hibák keresésére és javítására, így lehetővé téve a fejlesztőknek, hogy a termék fejlesztésére és új funkciók hozzáadására összpontosítsanak.

Mi az a kódbefedés

A kódbefedés (code coverage) a szoftverfejlesztés egyik alapvető metrikája, amely azt mutatja, hogy a forráskódból mennyi rész lett végrehajtva tesztelés során. Ez egy százalékos arányt jelent, amely megmutatja, hogy a kód mennyi részét futtattuk le legalább egyszer a tesztek alatt.

Miért fontos a kódbefedés?

A kódbefedés kulcsfontosságú azért, mert segít megérteni, hogy a tesztek mennyire alaposak és kiterjedtek. Magas kódbefedési arány arra utal, hogy a kód nagy része le lett tesztelve, ami javítja a fejlesztés minőségét és csökkenti a hibák lehetőségét. Emellett a kódbefedés segíthet azonosítani azokat a kódterületeket, amelyeket még nem teszteltek le, vagy amelyeket csak részben teszteltek.

Fontos megjegyezni, hogy a magas kódbefedés önmagában nem garantálja a jó minőségű szoftvert, mivel a tesztek minősége és azok lefedettségi szintje is meghatározó tényezők. Azonban a kódbefedés metrikája segíthet a fejlesztőknek és a csapatoknak abban, hogy megértsék, hol kell továbbfejleszteniük a teszteket, és hogy mely területeken kell nagyobb figyelmet fordítaniuk a kód minőségének javítása érdekében.

Típusai

Sor kódbefedés (Line coverage):

- A sor kódbefedés azt mutatja, hogy a tesztek mennyi százaléka futtatja le a forráskód sorait legalább egyszer.
- Ez a legegyszerűbb kódbefedés típus, és azt méri, hogy mely sorok kerültek végrehajtásra a tesztek során.

Elágazási kódbefedés (Branch coverage):

- Az elágazási kódbefedés azt méri, hogy a kód minden elágazása (if, switch, stb.) hányszor lett végrehajtva a tesztek alatt.
- Ez a típus mélyebb betekintést nyújt a kód különböző ágainak tesztelésébe, és segít az elágazásokból adódó lehetséges hibák felderítésében.

Ág kódbefedés (Condition coverage):

- Az ág kódbefedés azt mutatja, hogy a kód minden feltételének (logikai kifejezések) minden lehetséges értékét legalább egyszer tesztelték-e.
- Ez a típus még részletesebb képet nyújt a kód lefedettségéről, mivel figyelembe veszi az egyes feltételek különböző értékeit.

Funkcionális kódbefedés (Function coverage):

- A funkcionális kódbefedés azt méri, hogy a kód mennyi százalékát futott le a tesztek alatt funkciók szerint.
- Ez hasznos lehet olyan esetekben, amikor a kódbázis nagy része modulárisan van felépítve, és a funkciók külön-külön tesztelhetők.

Előnyei és kihívásai

Előnyök:

Jobb minőségű kód: A magas kódbefedési arány segít a hibák korai felderítésében és a kód minőségének javításában, mivel azok a területek, amelyeket nem teszteltek le, kiemelkednek, és így a fejlesztőknek lehetősége van javítani azokat.

Bizalmat adó tesztlefedettség: Magas kódbefedési arány növeli a fejlesztők és az érdekelt felek bizalmát a kód minőségével kapcsolatban, mivel azt jelzi, hogy a kód nagy része le lett tesztelve, és valószínűleg megbízható.

Hatékonyabb hibajavítás: A kódbefedés segít azonosítani azokat a területeket, amelyeket érdemes megvizsgálni hibákért, így hatékonyabbá teszi a hibajavítási folyamatot.

Tesztelési hatékonyság növelése: A kódbefedés metrikája segíthet a tesztelőknek abban, hogy jobban megértsék, hogy mely területeket kell még tesztelni, így hatékonyabbá téve a tesztek tervezését és végrehajtását.

Kihívások:

Túlzott függés a kódbefedésre: A kódbefedés metrikája önmagában nem jelent teljes biztonságot a kód minőségével kapcsolatban. A túlzott fókusz a kódbefedési arányra arra

késztetheti a fejlesztőket, hogy a kód lefedettségét maximalizálják, anélkül hogy figyelembe vennék a tesztek minőségét vagy relevanciáját.

"Fehér foltok" a tesztelésben: Az összes kódterület teljes lefedése nem mindig lehetséges vagy gyakorlati, különösen olyan esetekben, amikor ritka vagy nehézkes a tesztelés. Ez azt jelenti, hogy lehetnek olyan területek, amelyeket nem teszteltek le, és amelyek potenciálisan hibásak lehetnek.

Tesztelési erőforrások: Nagy kódbázisú projektek esetén a teljes kódbefedési arány elérése időigényes lehet, és nagy tesztelési erőforrásokat igényelhet.

Hamis biztonságérzet: Magas kódbefedési arány esetén a fejlesztők könnyen abba a csapdába eshetnek, hogy túlzottan biztonságban érzik magukat a kód minőségével kapcsolatban, és esetleg figyelmen kívül hagyják az egyéb minőségellenőrzési szempontokat.

Hogyan érjünk el magas kódlefedést

Tesztelési tervezés: Alapos teszttervezés segítségével megtervezhetjük a teszteket úgy, hogy lefedjék a kódbázis nagy részét. Ennek részeként azonosítsuk és priorizáljuk azokat a kritikus funkciókat és területeket, amelyeket tesztelni kell.

Tesztautomatizálás: A tesztek automatizálása lehetővé teszi, hogy gyorsabban és hatékonyabban futtassuk a teszteket, és így könnyebben elérjük a magas kódbefedést.

Tesztelési keretrendszerek használata: A tesztelési keretrendszerek, például a JUnit vagy a pytest, segítenek az egység-, integrációs és elfogadási tesztek létrehozásában és végrehajtásában, ami elősegítheti a magas kódbefedést.

Tesztpiramis alkalmazása: A tesztpiramis elve szerint a teszteknek különböző szinteken (egység-, integrációs, elfogadási) kell eloszlaniuk, és az egységteszteknek kell a legnagyobb arányban részt venniük a teljes tesztelésben. Ez segíthet elkerülni a túlzott tesztelési erőforrásokat, miközben biztosítja a megfelelő kódbefedést.

Tesztkorpusz bővítése: A meglévő tesztesetek kiegészítése új esetekkel és szcenáriókkal növelheti a tesztek lefedettségét, különösen olyan területeken, amelyek korábban nem voltak megfelelően lefedettek.

Refaktorálás és egyszerűsítés: A kód egyszerűsítése és a felesleges kód eltávolítása segíthet a tesztek készítésében és a kódbefedés növelésében, mivel a kevesebb és egyszerűbb kód könnyebben tesztelhető és karbantartható.

Gcov

A gcov egy tesztlefedettségi program. A GCC-vel együtt használhatod programjaid elemzésére, hogy segítsen hatékonyabb, gyorsabban futó kódot létrehozni, és felfedezze a program nem tesztelt részeit. A gcov-ot profiling eszközként is használhatod, hogy segítsen felfedezni, hogy az optimalizálási erőfeszítések hol hatnak a legjobban a kódodra. A gcov-ot a másik profiling

eszközzel, a gprof-fal együtt is használhatod, hogy felmérje, a kódodnak mely részei használják a legtöbb számítási időt.

Profiling

A profiling eszközök segítenek elemezni a kód teljesítményét. A gcov-hoz vagy a gprof-hoz hasonló profiling segítségével megtudhatsz néhány alapvető teljesítménystatisztikát, például:

- milyen gyakran hajtódnak végre az egyes kódsorok
- milyen kódsorok kerülnek ténylegesen végrehajtásra
- mennyi számítási időt használnak fel az egyes kódrészek

Ha már tudod ezeket a dolgokat arról, hogyan működik a kódod fordítás közben, megnézheted az egyes modulokat, hogy melyik modulokat kell optimalizálni. A gcov segít meghatározni, hol kell dolgozni az optimalizáláson.

A szoftverfejlesztők a lefedettségi tesztelést is használják a testsuite-okkal együtt, hogy megbizonyosodjanak arról, hogy a szoftver valóban elég jó a kiadáshoz. A tesztkészülékek ellenőrizhetik, hogy egy program az elvárásoknak megfelelően működik-e; a lefedettségi program azt vizsgálja, hogy a program mekkora részét gyakorolja a tesztkészülék. A fejlesztők ezután meghatározhatják, hogy milyen teszteseteket kell hozzáadni a tesztkészletekhez, hogy jobb tesztelést és jobb végterméket hozzanak létre.

A kódot optimalizálás nélkül kell lefordítanod, ha a gcov használatát tervezed, mert az optimalizálás, néhány kódsor egyetlen függvénybe történő egyesítésével, nem ad annyi információt, amennyire szükséged van ahhoz, hogy megkeresd azokat a "forró pontokat", ahol a kód sok számítógépes időt használ. Hasonlóképpen, mivel a gcov a statisztikákat soronként (a legalacsonyabb felbontásban) halmozza fel, a legjobban olyan programozási stílusban működik, amely minden sorba csak egy utasítást helyez. Ha bonyolult makrókat használsz, amelyek ciklusokra vagy más vezérlési struktúrákra bővülnek, a statisztikák kevésbé hasznosak - csak arról a sorról adnak jelentést, ahol a makróhívás megjelenik.

A gcov létrehoz egy sourcefile.gcov nevű naplófájlt, amely jelzi, hogy a sourcefile.c forrásfájl egyes sorai hányszor hajtódtak végre. Ezeket a naplófájlokat a gprof segítségével együtt használhatod a programjaid teljesítményének finomhangolásához. A gprof időzítési információkat ad, amelyeket a gcov-tól kapott információkkal együtt használhatsz.

A gcov csak a GCC-vel fordított kódokon működik. Nem kompatibilis semmilyen más profiling vagy tesztlefedettségi mechanizmussal.

Gcov GCC optimalizálással

Ha a gcov segítségével kívánja optimalizálni a kódot, akkor először a programot a GCC '--coverage' speciális opciójával kell lefordítania. Ettől eltekintve bármilyen más GCC-opciót használhatsz; de ha bizonyítani akarod, hogy a programod minden egyes sora végrehajtásra került, akkor nem szabad egyszerre fordítanod az optimalizálással. Egyes gépeken az optimalizáló képes egyes egyszerű kódsorokat más sorokkal kombinálva kiküszöbölni. Például az ilyen kódot:

```
if (a != b)
  c = 1;
else
  c = 0;
```

egyes gépeken egyetlen utasítássá fordítható. Ebben az esetben a gcov nem tudja kiszámítani az egyes sorok külön-külön végrehajtását, mivel nincs külön kód az egyes sorokhoz. Ezért a gcov kimenete így néz ki, ha a programot optimalizálással fordítottuk le:

```
      100:   12:if (a != b)
      100:   13:  c = 1;
      100:   14:else
      100:   15:  c = 0;
```

A kimenet azt mutatja, hogy ez az optimalizálással kombinált kódblokk 100-szor hajtódott végre. Bizonyos értelemben ez az eredmény helyes, mert mind a négy sort egyetlen utasítás képviselte. A kimenet azonban nem jelzi, hogy hányszor volt az eredmény 0 és hányszor volt az eredmény 1.

A hosszú futású alkalmazások használhatják az __gcov_reset és __gcov_dump lehetőségeket, hogy a profil gyűjtést az érdeklődésre számot tartó programterületre korlátozzák. Az __gcov_reset(void) hívása az összes futásidejű profil számlálót nullára törli, az __gcov_dump(void) hívása pedig az adott ponton összegyűjtött profil információkat a .gcda kimeneti fájlokba kiírja. A műszeres alkalmazások egy 99-es prioritású statikus destruktort használnak az __gcov_dump függvény meghívására. Így az __gcov_dump az összes felhasználó által definiált statikus destruktor után kerül végrehajtásra, mint a