GDBによるデバッグ

2020年10月16日（金）
システム開発プロジェクト応用第一

東京工業大学
特任助教　内田公太

本講義の概要

- 実際のシステム開発プロジェクトの現場で使われている現代的な開発ツールや手法を学ぶ
  - 正しいツールや手法の選択はソフトウェア開発を効率的に，そして楽しいものにする

到達目標：

- 現代的な開発ツールの基本的な使い方と適する用途が分かる

講義で扱うトピック

- 情報収集
- GDB
- Git
- バグトラッキング
- GitHub & Pull Request
- ユニットテスト
- 継続的インテグレーション
- デプロイと冪等性
- コミュニケーション

自己紹介

- 内田公太
- Twitter @uchan_nos
- 週3日：サイボウズ・ラボ株式会社
  週2日：東工大の特任助教

- osdev-jpコアメンバー
- 『30日でできる! OS自作入門』の校正担当
- 『自作エミュレータで学ぶ
  x86アーキテクチャ』の著者 →

講義の進め方

スタイル：

- 少し講義して演習，の繰り返し

成績評価：

- 現代の開発技術・手法の理解度を評価する
- 各トピックを受講者自身のソフトウェア開発プロジェクトに適用し，レポートおよびリポジトリを提出する
- レポートおよびリポジトリの充実度で成績を決定する

レポート，リポジトリの充実度

- 色々な要素がある
- トピックに対する回答
- ドキュメント
- コミットメッセージ
- プルリクのやり取り
- Etc.

- 総合的に判断して評価します

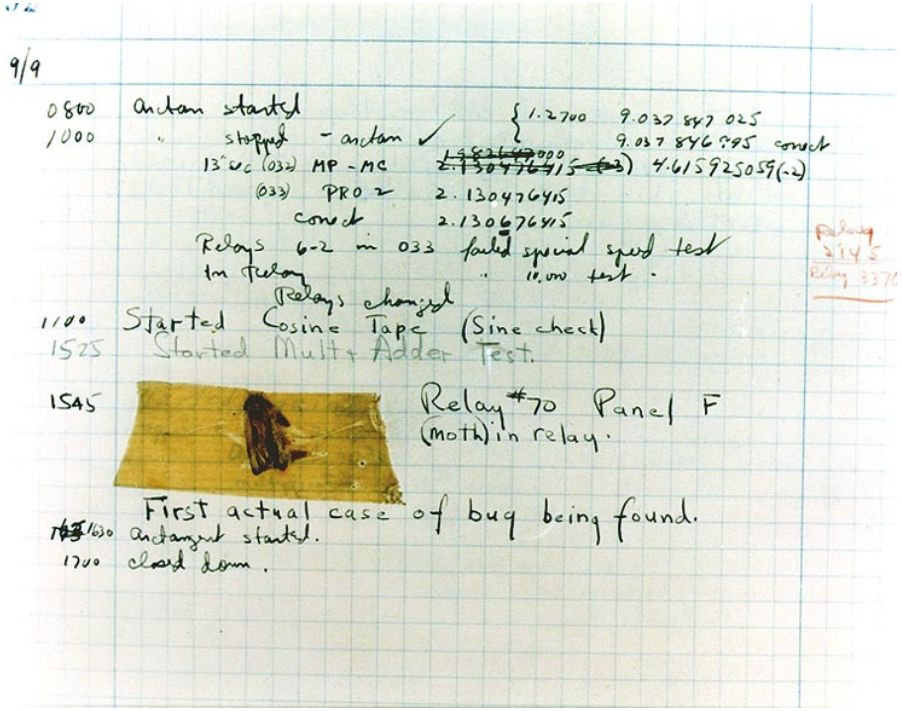
課題提出の方法

- 課題を含めたリポジトリとレポートを作成し，提出
- 前回（10/2）説明したので詳しい話はしないつもり
  - 改めて聞きたい方がいたらお知らせください

GDBによるデバッグ

プログラムのバグ

- バグ（Bug）＝意図せぬ不具合
  - コンピュータ登場以前から不具合を「バグ」と呼んでいたらしい
- 世界初のコンピュータバグ
  - 当時のコンピュータはリレー回路で，物理的接点があった



コンピュータの中に入りこんでいた「虫」の、おそらく最初の写真

https://ja.wikipedia.org/wiki/%E3%83%90%E3%82%B0

- 普通，仕様通りの挙動はバグとは言わない。「仕様です」
  - 仕様がおかしいことはよくある

デバッグする，とは

デバッグがソフトウェアのバグを取り除く過程だとしたら，プログラミングはバグを入れ込む過程に違いない

- Edsger W. Dijkstraの言葉とされる
  - https://www.goodreads.com/quotes/1271018
- ソフトウェアが期待通りに動くように，問題個所を特定して修正する行為

デバッグの色々な方法

- ソースコードを読み，アヤシイ個所を推測
- UBサニタイザを用いて未定義動作を検出
- printを仕込み，変数値などを確認（printデバッグ）
- デバッガで動作を追い，変数値などを確認
  - 今日の主題
- エスパーで問題を特定
  - 半分ネタ，半分ホント

UBサニタイザ

- Sanitizer：消毒剤
- Undefined Behavior Sanitizer：UBを検知する
  - UB（未定義動作）：C/C++での重大なバグ
- コンパイル時に検査コードを埋め込み，実行時に検査
- ClangとGCCで利用可

- Googleが開発しているサニタイザシリーズ
- 他にAddress/Leak/Thread/Memory Sanitizers

UB（未定義動作）

- 規格で挙動が規定されていない動作のこと
  - 例えばヌルポインタ参照，配列外参照，無限ループ
- UBは重大なバグ。プログラムにUBがあると，例えば
  - printfを埋め込んだだけで挙動が大きく変わる
  - 変数に書き込んだ値と違う値が読み出される
  - 鼻から悪魔が出る
- など，予想できない挙動となることがある

- →printfデバッグ等をする前にUBをなるべく減らそう

演習1

自分のOSSプロダクトを
UBサニタイザで検査してみよ

- C/C++で作ったソフトウェアが無ければ簡易なものを今作る
- clang -fsanitize=undefined target.c
- 制限時間10分

UBサニタイザの検知例

```
#include <stdio.h>
#include <string.h>
int main(int argc, char** argv) {
  if (strcmp(argv[1], "foo") == 0) {
    printf("foo!¥n");
  } else {
    printf("hello¥n");
  }
}
```

```
target.c:4:14: runtime error: null pointer passed as argument 1, which is declared
to never be null
/usr/include/string.h:137:33: note: nonnull attribute specified here
UndefinedBehaviorSanitizer:DEADLYSIGNAL
==96==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x000000000000 (pc
 0x7ff473ea9e8a bp 0x000000427e80 sp 0x7fffe553fc28 T96)
==96==The signal is caused by a READ memory access.
==96==Hint: address points to the zero page.
    #0 0x7ff473ea9e89  /build/glibc-OTsEL5/glibc-2.27/string/../sysdeps/x86_64/mult
iarch/strcmp-sse2-unaligned.S:30
```

GDB (GNU Debugger)

- Unix系システムの代表的なデバッガ
  - C/C++, Go, Objective-C, Rust等をサポート

- ブレークポイント，ステップ実行
- 変数値の確認，メモリダンプ

- $ gdb a.out（新たにプロセスを立ち上げる）
- $ gdb --pid=PID（既存プロセスに接続）

ブレークポイント

- プログラムを一時停止する箇所
  - 関数の先頭
  - プログラム行
  - 機械語

```
auto it = table.find(".bss");
if (it == table.end()) {
  return;
}
```

- ブレークポイントで停止させ，その時の状態を調査
  - 変数値を確認したり，メモリダンプしたり

- bコマンドで設定
  - `(gdb) b main`
    `Breakpoint 1 at 0x400531: file main.cpp, line 14.`

GDBを使う準備

- 対象ソフトウェアをデバッグ情報付きでビルド
  - デバッグ情報により，シンボルがGDBから見え，デバッグしやすくなる
- Cコンパイラの -g オプション

```
$ clang -g target.c
```

- GDBの上で起動させる

```
$ gdb a.out
```

GDBの代表的なコマンド

コマンド	説明
run	プログラムを開始する
start	プログラムを開始し，main()の先頭で止まる
b main.cpp:42	main.cppの42行目にブレークポイントを設定する
b myfunc	myfunc()の先頭にブレークポイントを設定する
c	プログラムの実行を再開する
step	次の行まで実行する
bt	バックトレースを表示する
p foo	変数fooの値を表示
x/4gx $rsp	スタック値を8バイト×4個表示する

演習2

自分のOSSプロダクトに対し,
GDBでブレークポイントを設定して
変数値を確認せよ

- bでブレークポイントを設定しrunで実行
- ブレークしたらpで変数を表示

- 制限時間10分

デバッガによるデバッグのうれしさ

- コードを変更する必要が無い
- printデバッグより広範な情報を得られる
- ライブラリの中も探索できる
  - printは自分で書いたコードにしか埋め込めない
- コアダンプ解析ができる

コアダンプ

- プログラムのある時点のメモリ内容を記録したもの
- コアダンプから，その時点のプログラム状態を再現可能
- →プログラムクラッシュ時の原因解析に便利

```
$ clang -O3 -g -o target target.c
$ ./target
Segmentation fault (コアダンプ)
$ ls core
core
$
```

コアダンプの使い方

1.クラッシュ時にコアファイルが生成されるように設定

```
$ ulimit -c
0
$ ulimit -c unlimited
$ ulimit -c
unlimited
```

2.対象をクラッシュさせる

```
$ ./target
Segmentation fault (コアダンプ)
```

3.コアファイルを使ってGDBでデバッグする

```
$ gdb ./target core
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
…
Core was generated by `./target'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  __strcmp_sse2_unaligned () at …
(gdb)
```

演習3

コアダンプを利用したデバッグを試せ

- クラッシュするプログラムを作って試そう
  - ヌルポインタアクセス，配列外参照など
- 制限時間10分

ブレークポイントの仕組み

- ソフトウェア割り込みの仕組みを利用
  - ソフトウェア割り込みはCPUの機能
  - 事前に割り込みハンドラを設定する
- x86のint3命令は1バイト，任意の箇所に埋め込める
  - int3　　CC
  - int imm8　CD ib
- int3がBP例外を出す
  →TRAPシグナルが生成される
  →ptraceが捕捉し，デバッガに処理が渡る

int3命令の実験

```
$ clang -g -o int3 int3.c
$ ./int3
debug1
Trace/breakpoint trap (コアダンプ)
$
```

Int3命令でTRAPシグナルが発生する様子

```
#include <stdio.h>

int main() {
  printf("debug1¥n");
  asm("int3");
  printf("debug2¥n");
  printf("debug3¥n");
}
```

対象プログラム

int3命令でブレーク

プログラムの出力

埋め込んだint3による中断

ブレークポイントによる中断

```
$ gdb ./int3
…
(gdb) b int3.c:7
Breakpoint 1 at 0x4004fe: file int3.c, line 7.
(gdb) run
Starting program: /home/uchan/workspace/cpptest/int3
debug1

Program received signal SIGTRAP, Trace/breakpoint trap.
main () at int3.c:6
6         printf("debug2¥n");
(gdb) c
Continuing.
debug2

Breakpoint 1, main () at int3.c:7
7         printf("debug3¥n");
(gdb)
```

演習4

自分のOSSプロダクトにint3命令を
埋め込んでデバッグを実験せよ

- 制限時間10分

ウォッチポイント

- 変数の値変化を監視する機能
  - ウォッチポイントはデータが対象
  - ブレークポイントはプログラムが対象
- 変数の値が変化したら実行が中断される

```
$ gdb ./watch
…
(gdb) start 3 a b 5
…
(gdb) watch sum
Hardware watchpoint 2: sum
(gdb) c
Continuing.
…
```

```
int main(int argc, char** argv) {
  int sum = 0;
  for (int i = 1; i < argc; ++i) {
    if ('0' <= argv[i][0] &&
        argv[i][0] <= '9') {
      sum += argv[i][0] - '0';
    }
  }
  return sum;
}
```

対象プログラム

ウォッチポイントの実験

```
…
Hardware watchpoint 2: sum

Old value = 0
New value = 3
main (argc=5, argv=0x7fffffffdf88) at watch.c:7
7         }
(gdb) p i
$1 = 1
(gdb) c
Continuing.

Hardware watchpoint 2: sum

Old value = 3
New value = 8
main (argc=5, argv=0x7fffffffdf88) at watch.c:7
7         }
(gdb) p i
$2 = 4
(gdb) c
```

```
int main(int argc, char** argv) {
  int sum = 0;
  for (int i = 1; i < argc; ++i) {
    if ('0' <= argv[i][0] &&
        argv[i][0] <= '9') {
      sum += argv[i][0] - '0';
    }
  }
  return sum;
}
```

対象プログラム

ウォッチポイントの仕組み

Tokyo Tech

ハードウェアウォッチポイント

- x86ならデバッグレジスタ（DR0-DR7）
- CPUの機能としてメモリ読み書きを監視できる
  - 低オーバーヘッド

ソフトウェアウォッチポイント

- ハードウェア支援が使えない場合に選択される
- ソフトウェアでメモリ変更を監視
  - 高オーバーヘッド

参考 https://sourceware.org/gdb/wiki/Internals%20Watchpoints

デバッグレジスタ

監視設定

ステータス

監視対象の
メモリアドレス

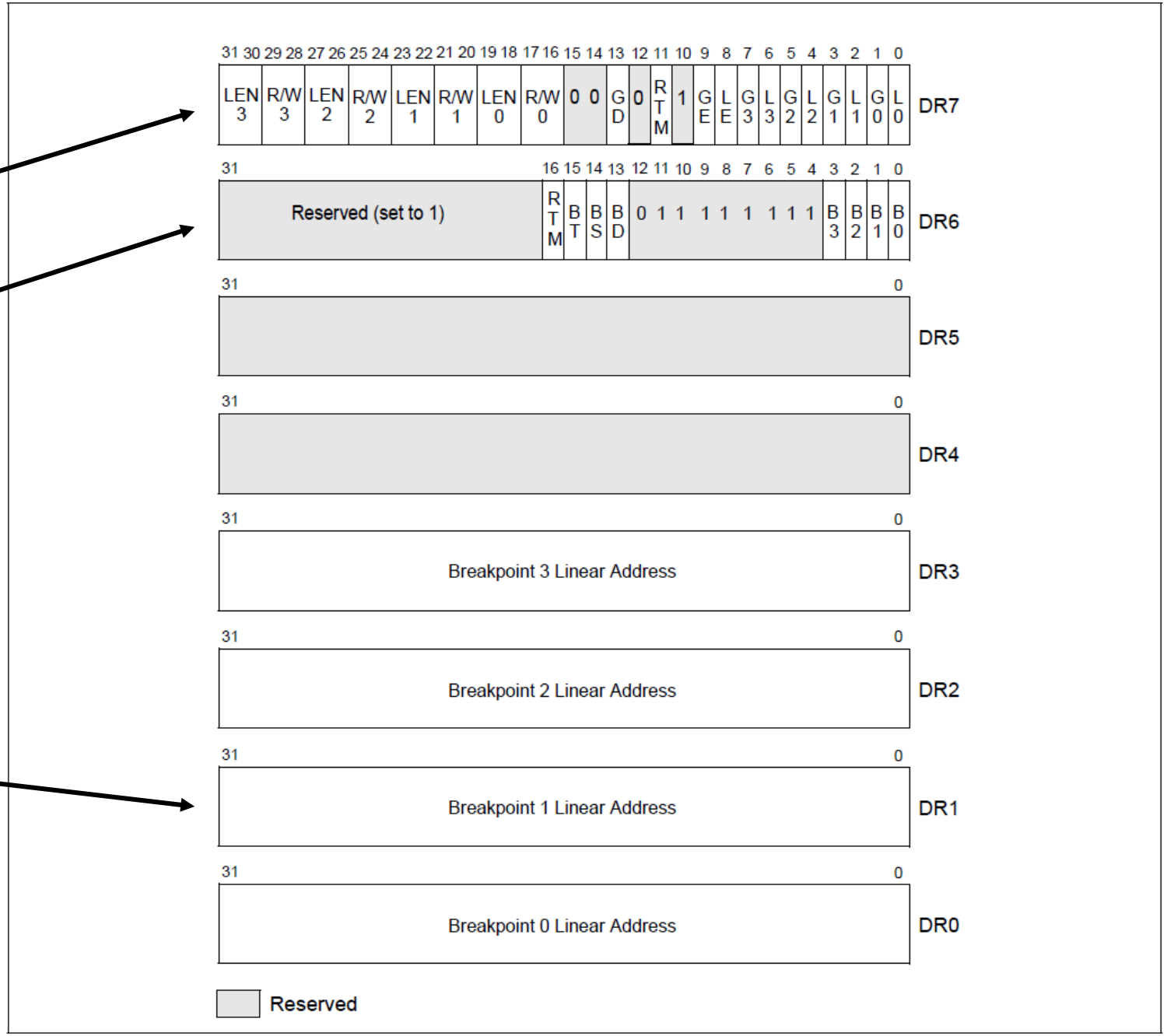


Figure 17-1. Debug Registers

右図はIntel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 より引用

演習5

自分のOSSプロダクトの変数変化を
ウォッチポイントを用いて観察せよ

- ソフトウェアウォッチポイントが使われる条件は何だろう？
- 制限時間10分

演習6

自分のOSSプロダクトのデバッグを
作業記録を取りながら行おう

- デバッグだけでなく開発作業を並行にやってもOK
  - デバッグは開発作業に伴って必要なものだから
- どんなデバッグ手法を使ってもOK
  - 作業記録にはもちろん用いた手順を記録しよう

課題提出

- 課題を含めたご自身のリポジトリとレポートを提出
- 提出先は内田のGitHubリポジトリ
  - https://github.com/uchan-nos/titech-sysdev-2020
  - プライベートリポジトリのためアカウント登録必須
    皆さんのGitHubアカウントを教えてください
- このリポジトリに対し，レポートを送る
  - レポートには，トピックに対する回答を含める
- 提出期限は講義の1週間後の10:00（JST）
  - 「情報収集」の課題提出期限は10/09（金）10:00

アカウント申請

- GitHubアカウントを内田に教える作業（初回のみ）
- 情報を uchida.k.afあっとm.titech.ac.jp に送る
- メールに含める内容
  - GitHubアカウント名
  - 本名
  - 学籍番号

- 今作業してください
- ここで，皆さんからアカウント情報が来るのを待つ

レポートの送り方

1. 独自のブランチを作る
   1. titech-sysdev-2020:master
      ↓branch
      titech-sysdev-2020:report-YOUR_NAME
2. 回答の概要をまとめたファイルを加える
   1. titech-sysdev-2020/reports/TOPIC/YOUR_NAME.md
   2. Commit & Push
3. プルリクを送る（リポジトリ内プルリク）
   1. titech-sysdev-2020:report-YOUR_NAME
      ↓pull request
      titech-sysdev-2020:master

レポート

- reports/TOPIC/YOUR_NAME.md
- このファイルに課題への回答を記載する
- 必要なら以下のものを含める
  - Issueへのリンク
  - コミット差分へのリンク
    https://github.com/HOGE/REPO/compare/COMMIT1...COMMIT2
  - その他
- 要するに，成績評価に必要な情報をYOUR_NAME.md自体に記載するか，そこから辿れるようにする

他のファイルを含める場合

- YOUR_NAMEディレクトリにファイルをまとめる

- Example:

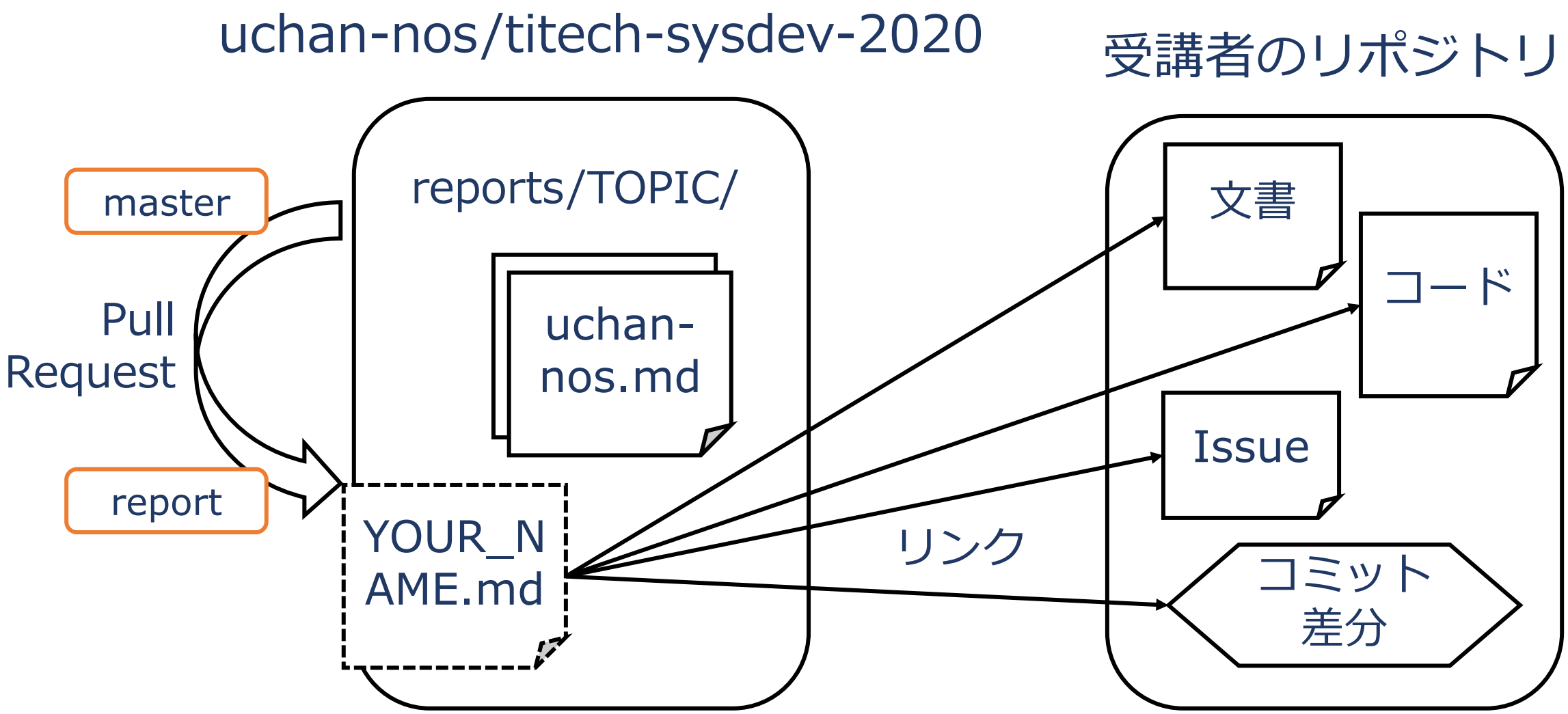
```
reports/TOPIC/
  YOUR_NAME.md
reports/TOPIC/YOUR_NAME/
  foo.js
  bar.css
  baz.html
```

良い作業記録

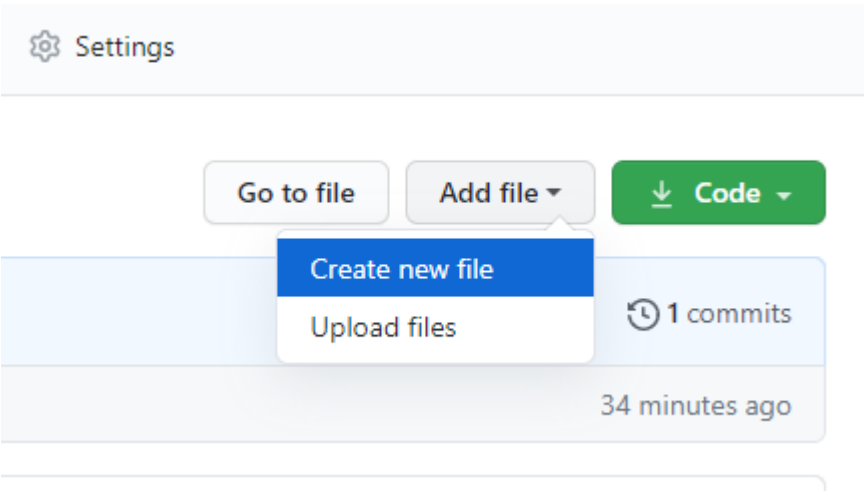
- 実行したコマンドが正確に記録されている
- 参照したサイトのURLや文献名が正確に記録されている
- 手順に抜けが無い
- 重要な処理結果が載っている

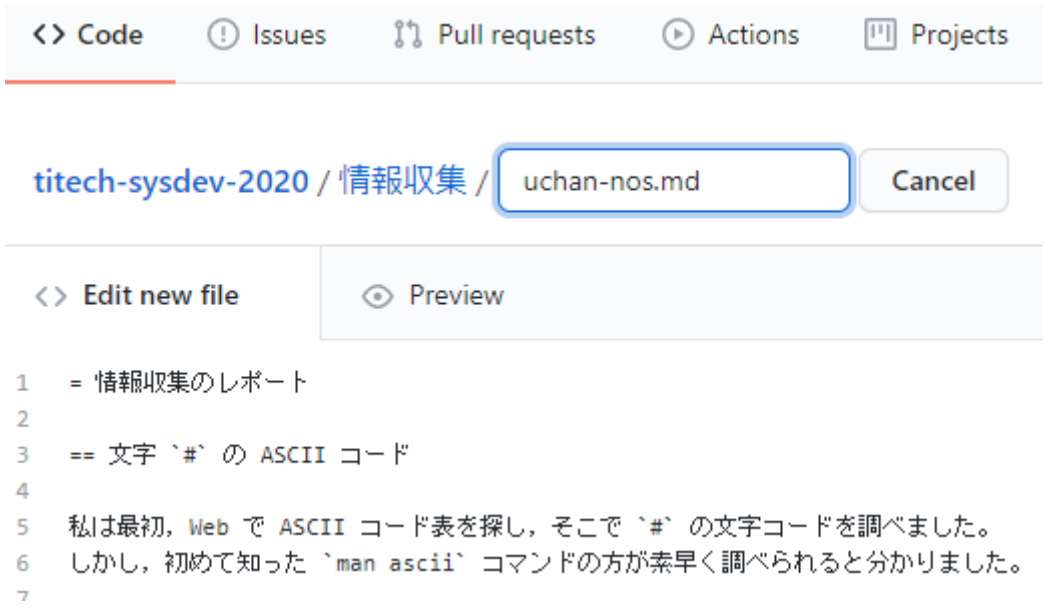
- 要するに：
  見返したときに，何を実行して何が起きたか分かる

レポートの構成


uchan-nos/titech-sysdev-2020
受講者のリポジトリ
master
Pull Request
report
reports/TOPIC/
uchan-nos.md
YOUR_NAME.md
リンク
文書
コード
Issue
コミット差分

レポートの送り方 1/2

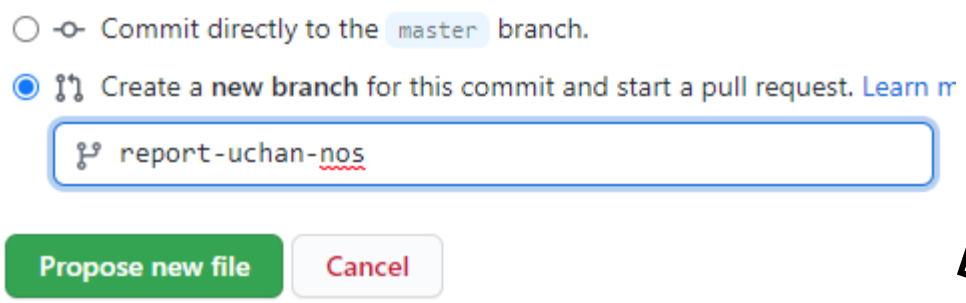




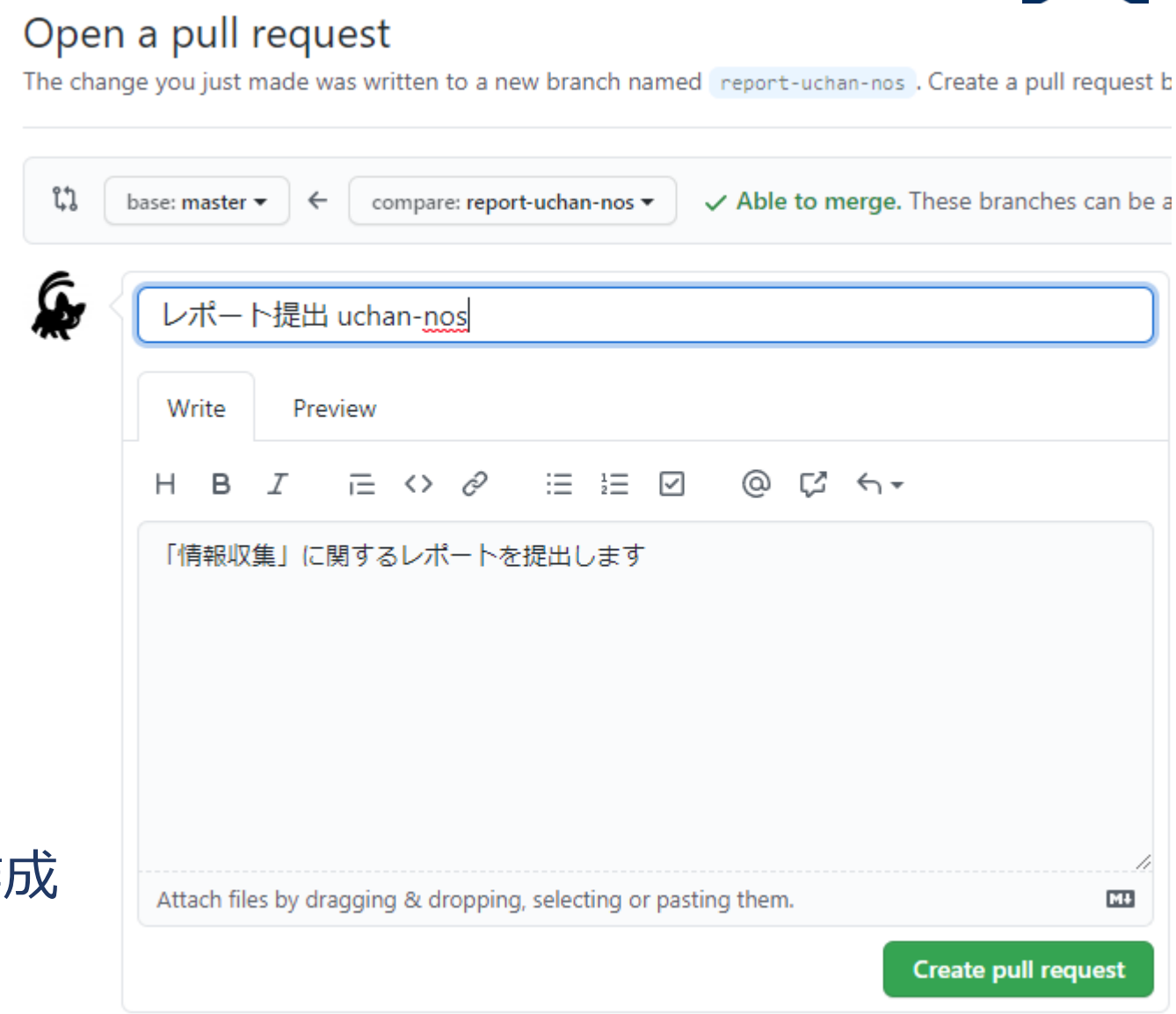
ファイルを新規作成

YOUR_NAME.mdの内容を記述

レポートの送り方 2/2

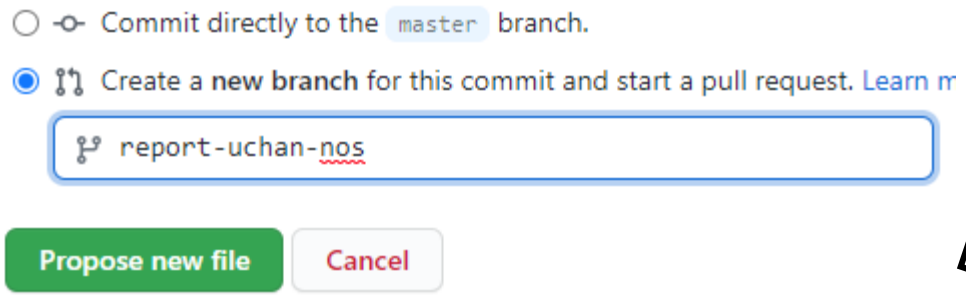


新規ブランチにコミット

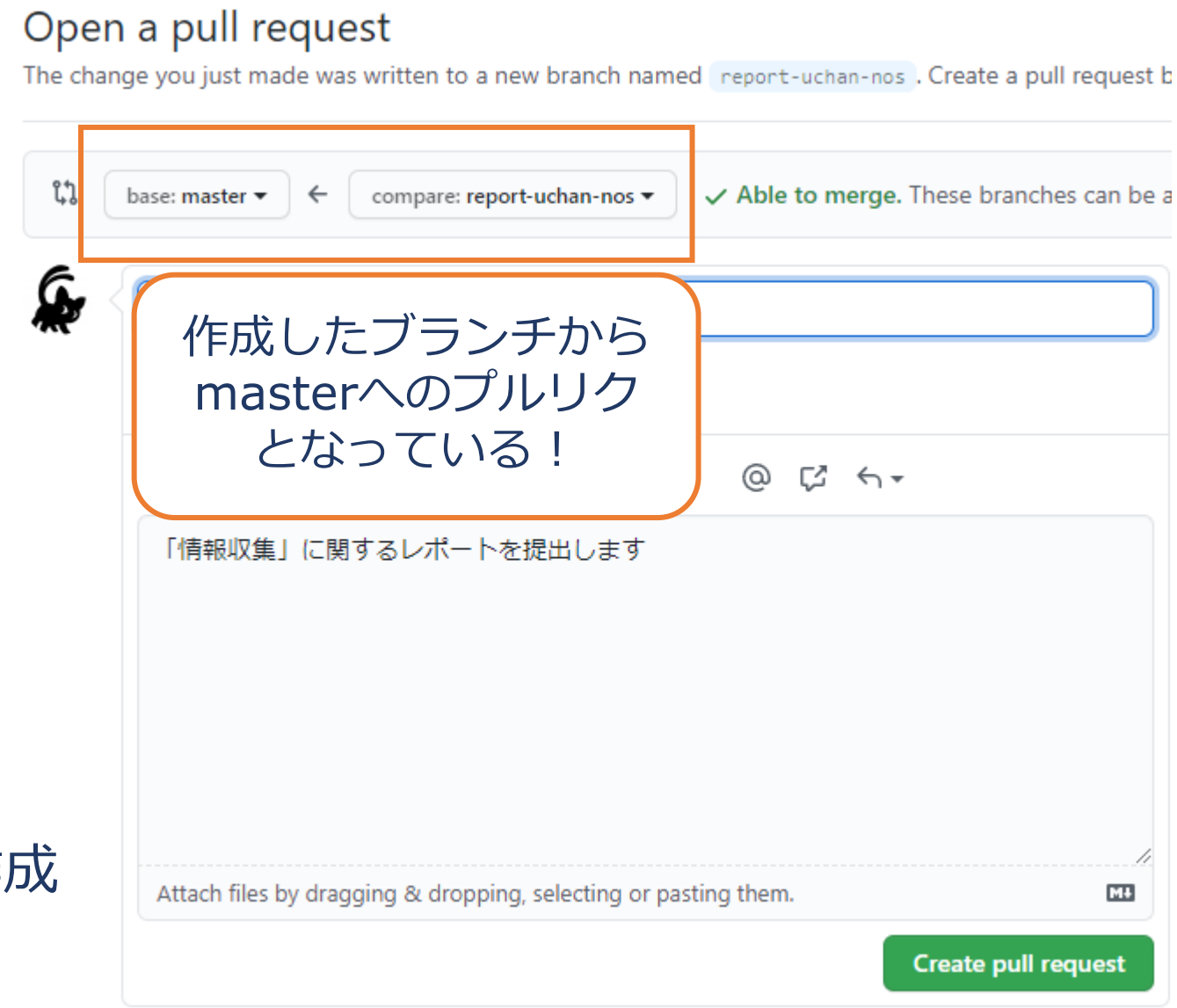


プルリクを作成

レポートの送り方 2/2



新規ブランチにコミット



作成したブランチから
masterへのプルリク
となっている！

プルリクを作成

レポートの受理

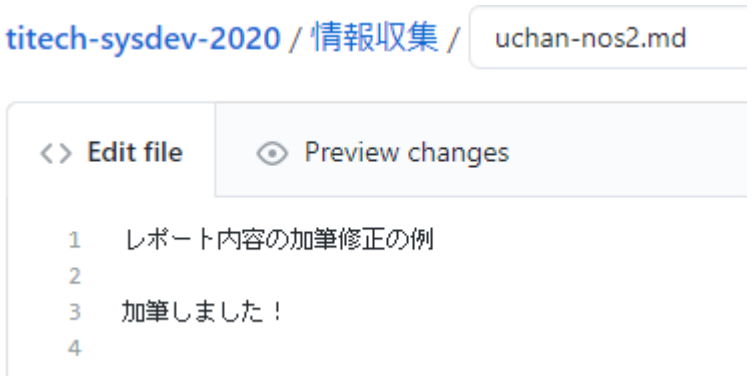
プルリクにコメントが付くことも

Merged：レポート受理済み

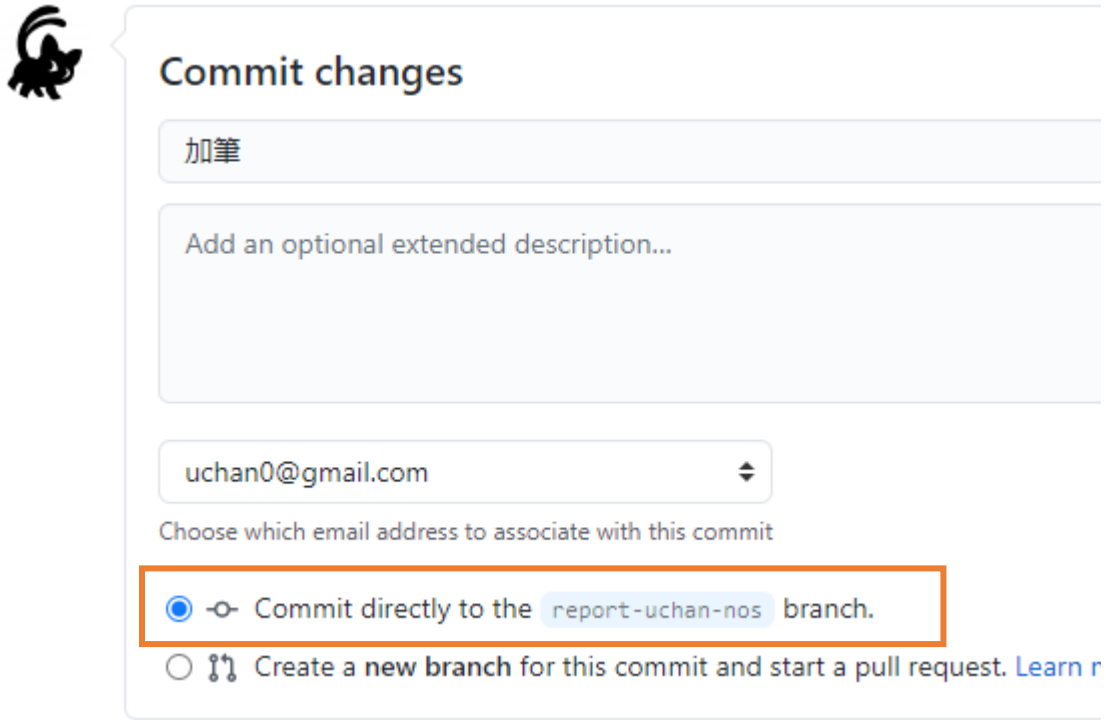
レポートの更新 1/2

レポートに不十分な個所があった！

まだマージされてないときの
更新方法を紹介



納得いくまで加筆修正



report-Xブランチにコミット

レポートの更新 2/2

一発目のコミット

Pull Requestに
コミットが追加されていく

レポートの送り方：その他の方法

- GitHubのWebインターフェースを使う必然性はない
- コマンドラインで作業してもよい
  - 具体的なコマンドラインは示しません
  - この講義は「情報収集」でしたね？
  - コマンドラインについて情報収集すれば，レポートをさらに充実させるネタになりますよ

次回予告

- 次回は10/23（金）14:20から
- 「Gitによるバージョン管理」をやります
- 手元でgitコマンドを使えるようにしておいてください

```
uchan@workstation:~/workspace/myproj$ git version
git version 2.17.0
uchan@workstation:~/workspace/myproj$
```