【Git】状態管理の概念と変更の保存

Gitで管理する３つの状態

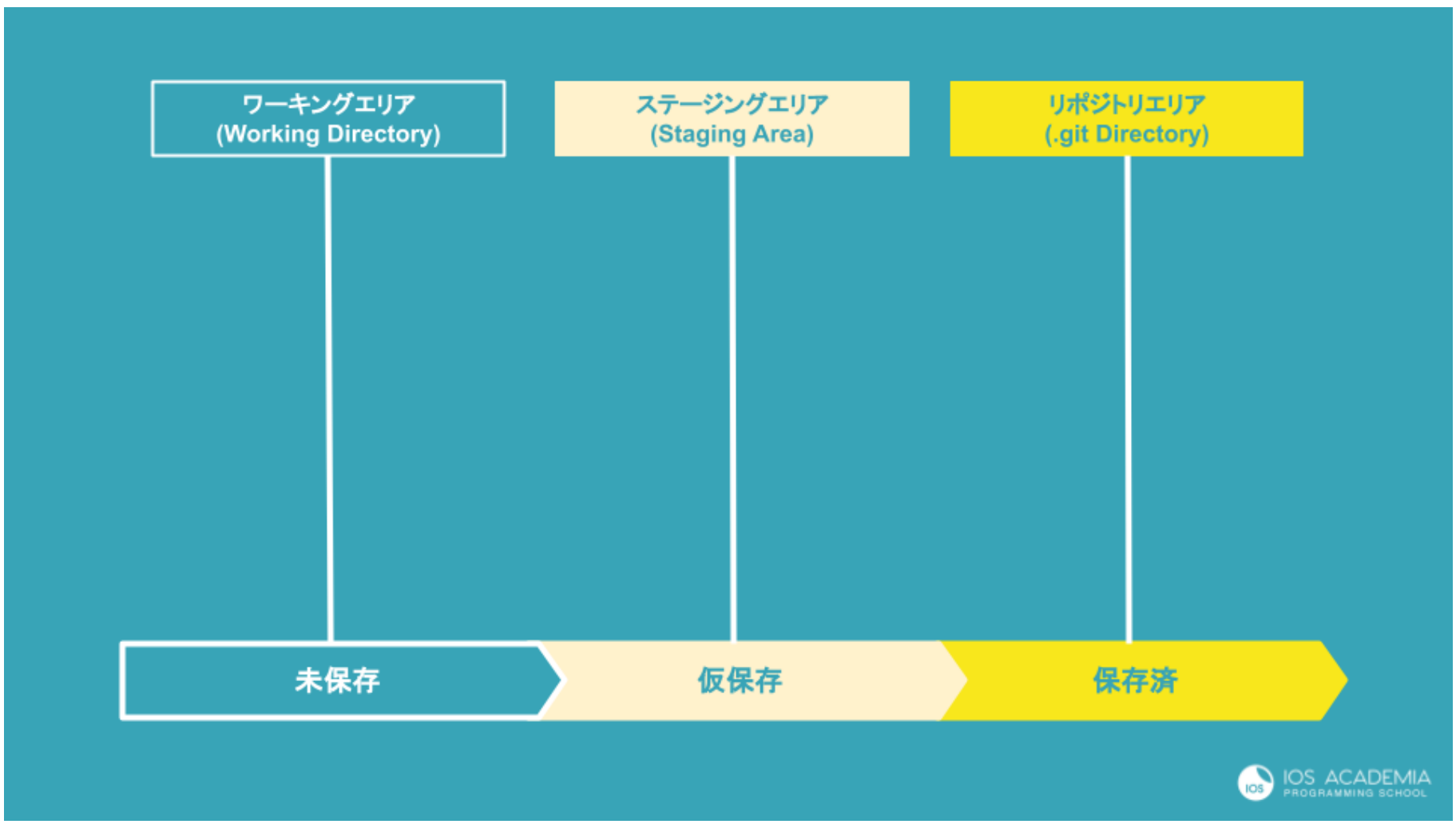
Gitを使ってソースコードの管理を行う場合には、主に3つの状態が存在します。

1. 未保存状態(Working Directory)
2. 仮保存状態(Stagin Area)
3. 保存済状態(.git directory)

Gitで管理されているソースコードは、常にこの3つの状態のいずれかに存在しており、開発を進めて行く上では、最終的に全て「保存済状態」にすることで適切に管理を行うことが可能です。

ここからはそれぞれの状態を分かりすくするために、下記の様に呼びたいと思います。

1. ワーキングエリア(未保存)
2. ステージングエリア(仮保存)
3. リポジトリエリア(保存済)



新しいファイルを作成して状態を保存する

では実際にGitで管理を行ってるリポジトリ内でファイルを作成し、そのファイルをGit上に保存してみましょう。

新しいファイルを作成

まずは現在のリポジトリがどこにあるのか確認しましょう。

```
command

$ pwd
```

```
command

$ /Users/yamataku/Git
```

現在は`yamataku`配下の`Git`というディレクトリにいることが分かります。

では次に今のディレクトリ配下のファイルとフォルダの確認をしてみます。

```
command

$ ls
```

すると、現状だと現在の`Git`ディレクトリ配下には(隠しファイル以外は)何のファイルも無いので、出力はありませんでした。

ではここに新しいファイルを作成していきましょう。

ファイル作成をする際には`touch`コマンドを使います。

```
command

$ touch first.txt
```

この様に`touch`の後に半角スペースを空けてファイル名を書くことで、新規ファイル作成ができます。

では現在作成したファイルを確認してみます。

```
command

$ ls
```

```
first.txt

hoge
```

そうすると今作成した`first.txt`が出力されるので、正常にファイルが作成できたことが分かりました。

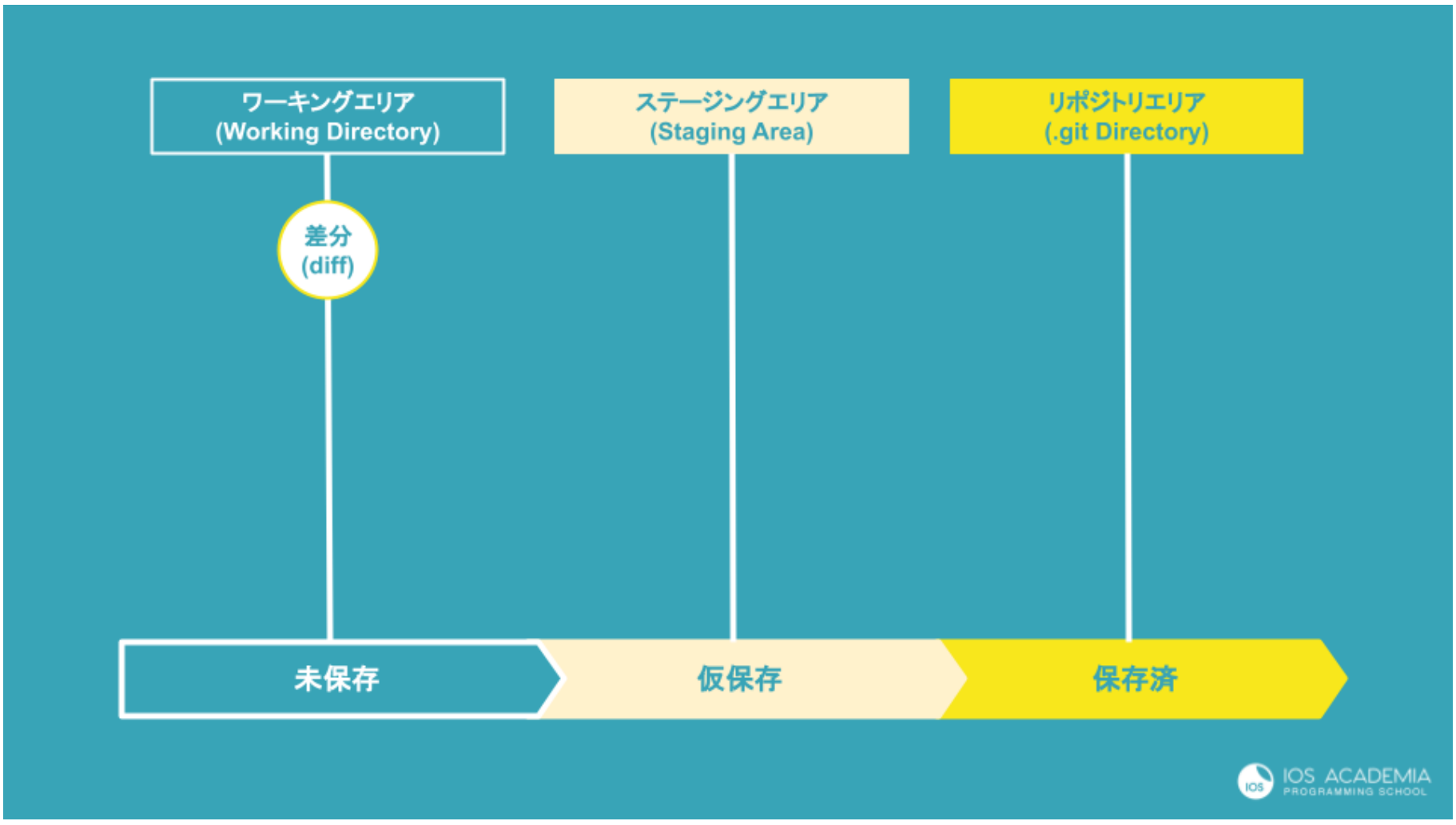
この時、ディレクトリをテキスト表すと下記のようになります。

```
directory

User
  └ yamataku
      └ Git
          └ first.txt
```

Git上で状態を確認

さて、ではここでGitでの管理状態を確認してみます。



Git管理上では、現在ワーキングエリア内に「未保存状態」の`first.txt`というファイルが追加されたことになっています。

この時、「`first.txt`が存在し無い状態」から「`first.txt`が存在する状態」に変化したため、「差分(diff)」が生じたことが分かります。

この差分(diff)はターミナル上でGitのコマンドを使って、具体的な内容を確認することが可能です。

command

```
$ git status
```

output

```
On branch master
No commits yet
Untracked files:
  (use "git add ..." to include in what will be committed)
    first.txt
nothing added to commit but untracked files present (use "git add" to track)
```

そうすると英語で差分の結果が表示されました。

Goole翻訳でも何でも構わないので、こうした英語の文章は必ず読んで内容を把握するようにしましょう。

(※英語を読む。というのは、エンジニアとしてとても重要な素養です。)

さて、ここでは何が書かれているかと言うと、`Untracked files`(≒ワーキングエリア)の中に`first.txt`と記載があるため、つまりは「first.txt」はまだ追跡されていない(≒ステージングエリアに仮保存されていない)という意味となります。

また、`use "git add <file>..." to include in what will be committed`と書いてありますが、これは`git add`というコマンドを使用して、コミット(≒リポジトリに保存)する対象ファイルに含めましょう。という意味になります。

つまり、現状だとワーキングエリア上で`first.txt`が「未保存」状態になっているため、`git add`というコマンドを使ってステージングエリアに「仮保存」しましょう。という指示が書かれているので、上述した図の内容になっている事が分かりました。

ステージングエリアに仮保存する

ではこの指示内容に従って、`first.txt`をステージングエリアに仮保存しましょう。
仮保存をする際には、`git add`コマンドを使うという指示があったので、それに従います。

```
command

$ git add first.txt
```

ステージングに状態を仮保存する場合は、この様に`git add`の後に半角スペースを空けて保存したいファイル名を指定することで、仮保存をすることができます。
では確認のためにもう一度状態を確認してみましょう。

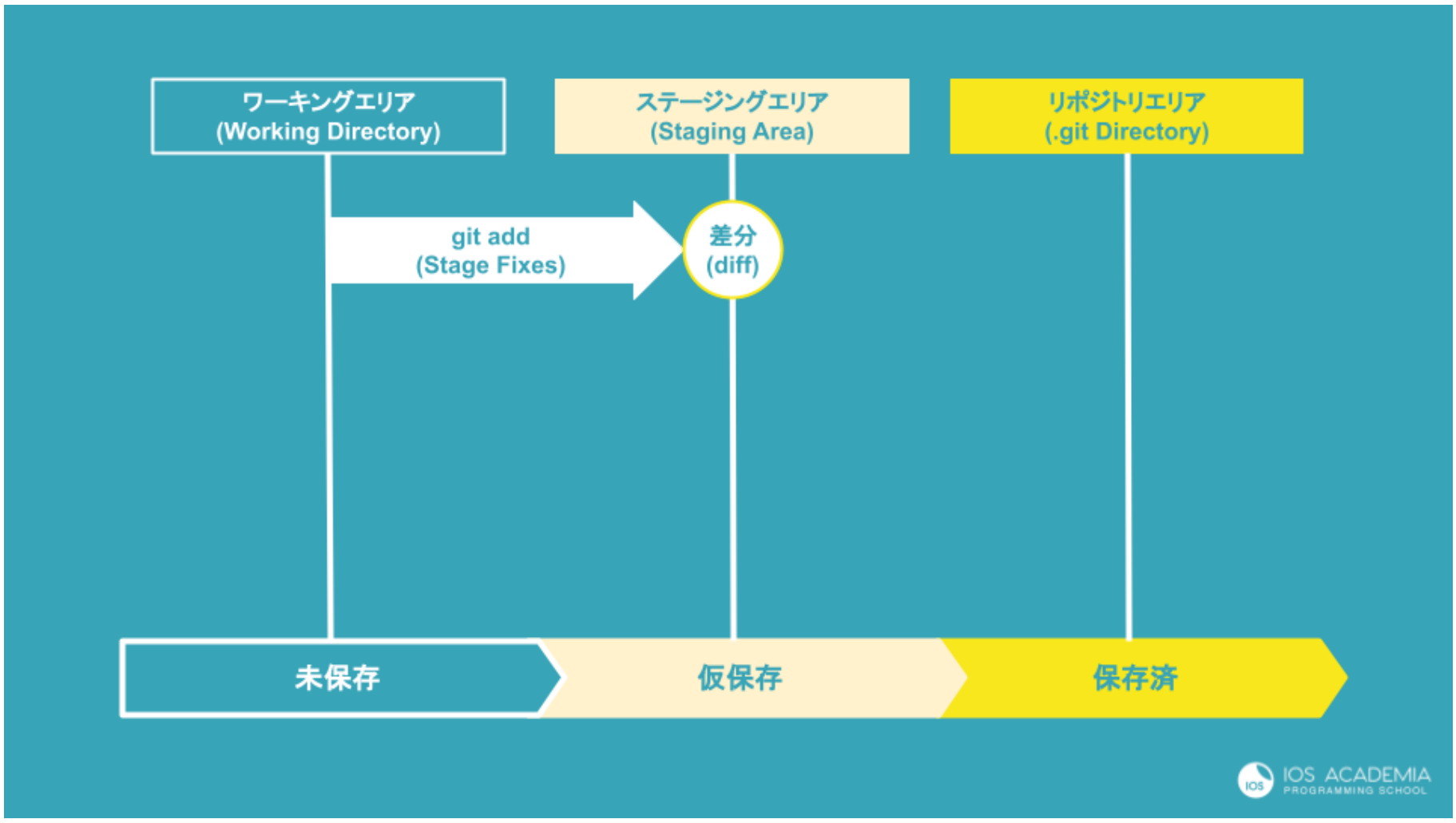
```
command

$ git status


output

On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached ..." to unstage)
    new file:   first.txt
```

そうすると、今度は出力結果が変わり、`new file: first.txt`と表示されたので、正常にファイルの仮保存がされたことが分かります。
先程の図でいうと、下記の様な状態になった訳ですね。



ですが、このままだとまだ`first.txt`はGit上のリポジトリエリアに「保存済状態」になっていません。

Gitの世界では、基本的に開発した内容を全て保存状態にすることで初めて正常に管理をすることができるため、このまま保存状態まで持っていきましょう。

コミットしてリポジトリエリアに状態を保存する

ステージングエリアに仮保存された差分は、`git commit`というコマンドを使用してリポジトリエリアに保存することができます。

```
command

$ git commit


vim

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   first.txt
#
.
.
.
"~/Git/.git/COMMIT_EDITMSG" 11L, 231C
```

そうすると、なにやらこれまでとは異なる出力がされました。

これは何が起きているかと言うと、「vim」というエディターが開いている状態となります。

エディターとは「編集・編集者」という意味ですが、ここではソースコードファイルを編集するためのもの。という理解をすれば問題ありません。

そして「vim」とは、CLI上で直接ファイルなどの編集をすることができるエディターツールのことで、これはmacOSに標準で用意されているものとなります。

vimの詳しい使い方は多くの記事で解説していますので、ここでは簡単な操作だけを説明したいと思います。

vimでコミットメッセージを書く

vim上では`Please enter the commit message for your changes.`と書かれているので、コミットする際のメッセージとなる文字列を登録するように指示が書かれています。

コミットメッセージ(`commit message`)とは、Git上で状態を保存した際に「その変更は一体何の内容が含まれているのか」というものを示す際に活用されます。

また、vimでは編集をする際にキーボード上にある英語キーの「i」をタップすることで、編集状態に入ることができるので、まずは「i」キーを押してコミットメッセージを書きましょう。

```
vim
```

```
first.txtを新規作成
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
#
# Changes to be committed:
#       new file:   first.txt
#
.
.
.
-- INSERT --
```

`first.txt`を新規作成という文字列を入力しておきました。

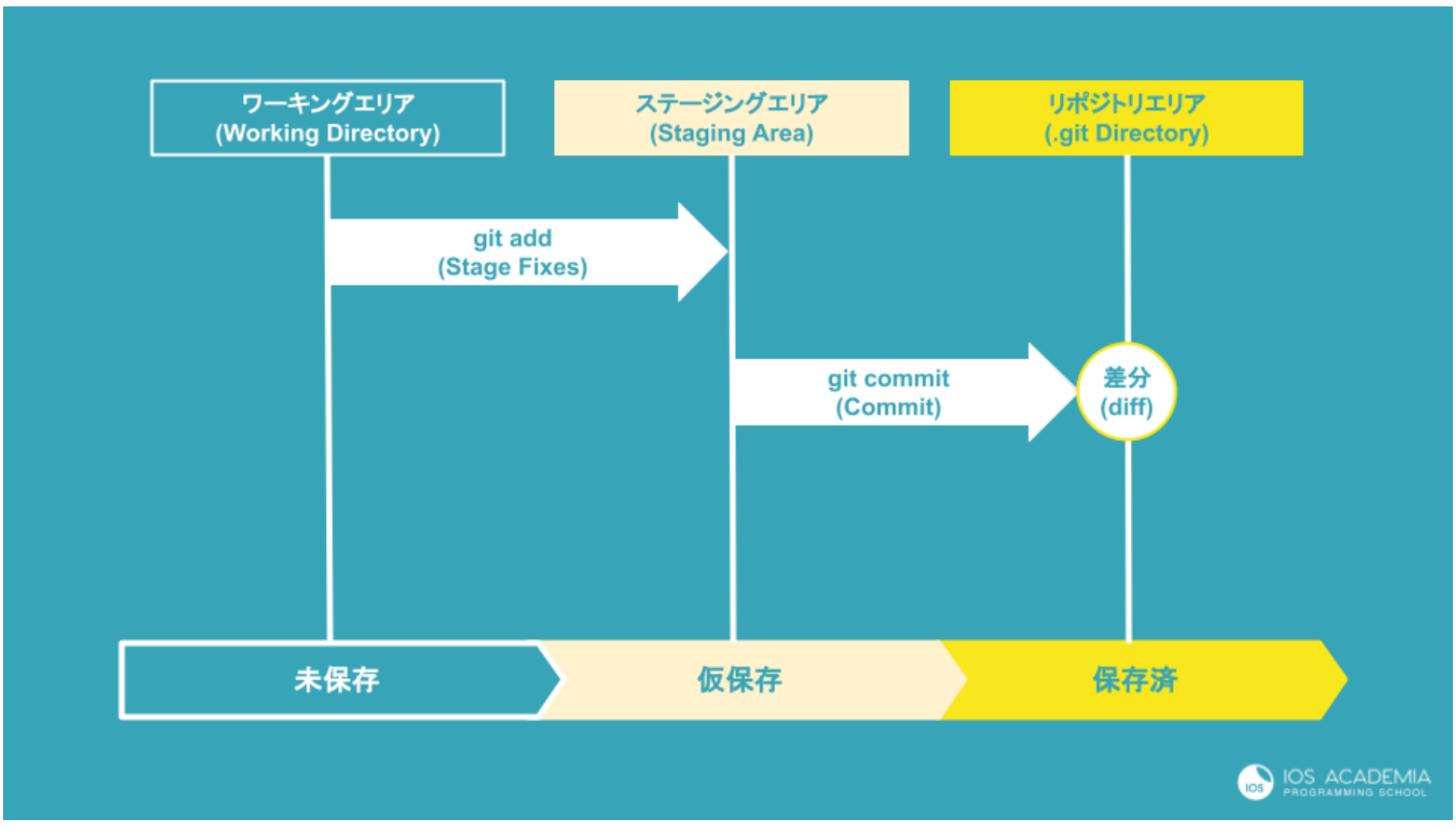
vimでは、編集状態にある時に`-- INSERT --`という表記に切り替わるので、この状態の時は編集中であると認識すればOKです。

では、vim上でこれを保存していきましょう。
保存をする場合は、キーボードのエスケープボタン(「esc」キー)で編集状態から抜け出せます。
(※エスケープすると`-- INSERT --`表記が消えるはずです)

そして保存をする場合は、大文字のゼットを2つ(`ZZ`)打てばOKです。
(※この時キーボードが日本語入力モードではなく、英数字入力モードになっていないと正常に機能しないので注意してください)

これで先ほどまでのターミナル表記に戻ったはずです。

ここまででようやく`first.txt`の内容をリポジトリエリアに保存する事ができました。
図の状態としては以下の通りです。



コミット履歴を確認する

この様にソースコードに変更を加えて、その差分をGit上でコミット(保存)することでソースコードを適切に管理しながら開発を進めることができます。

基本的には、今回の様な①ワーキングエリアに差分発生→②ステージングエリアに仮保存→③リポジトリエリアに保存、といった流れを繰り返すことでソースコードをどんどん改変していきますが、この改変の履歴を確認したい場合も出てきます。

その時には、`git log`というコマンドを使用して過去のコミット履歴の確認をすることも可能です。

```
command

$ git log
```

```
output

commit fcb0091678eb07ba90e8bcbd960abcbc080b528f (HEAD -> master)
Author: yamataku29
Date:   Thu Apr 8 21:21:30 2021 +0900
```

そうすると、上記の様な内容が出力されますが、それぞれの項目の意味は下記の通りとなっています。

- `commit`→コミットの「ハッシュ」(または「ハッシュ値」(※1)とも呼ばれる)
- `Author`→コミットを保存したGitアカウントのIDとメールアドレス
- `Date`→コミットをした際の日時

これからコミットを繰り返していくことで、上記のコミット履歴に表示される情報がどんどんと積み上がっていくことになります。
また、`git log`を実行した後にコミット履歴を閉じたい場合は、キーボードにある英語の「q」を押せば元のターミナル表記に切り替えることができます。

注釈

- ※1: 差分を保存したコミット自体を一意に識別するために発行された英数字の乱数のこと

← 教材一覧へ戻る

受講申し込みはこちら